European XFEL Python data tools Documentation

Release 0.7.0

European XFEL

Dec 20, 2019

CONTENTS

karabo_data is a Python library for accessing and working with data produced at European XFEL.

CHAPTER

ONE

INSTALLATION

karabo_data is available on our Anaconda installation on the Maxwell cluster:

```
module load exfel exfel_anaconda3
```

You can also install it from PyPI to use in other environments with Python 3.5 or later:

```
pip install karabo_data
```

If you get a permissions error, add the `--user` flag to that command.

CHAPTER

TWO

QUICKSTART

Open a run or a file - see *Opening files* for more:

```
from karabo_data import open_run, RunDirectory, H5File

# Find a run on the Maxwell cluster
run = open_run(proposal=700000, run=1)

# Open a run with a directory path
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0001")

# Open an individual file
file = H5File("RAW-R0017-DA01-S00000.h5")
```

After this step, you'll use the same methods to get data whether you opened a run or a file.

Load data into memory - see *Getting data by source & key* for more:

```
# Get a labelled array
arr = run.get_array("SA3_XTD10_PES/ADC/1:network", "digitizers.channel_4_A.raw.samples
↪")

# Get a pandas dataframe of 1D fields
df = run.get_dataframe(fields=[
    ("*_XGM/*", "*.i[xy]Pos"),
    ("*_XGM/*", "*.photonFlux")
])
```

Iterate through data for each pulse train - see *Getting data by train* for more:

```
for train_id, data in run.select("*/DET/*", "image.data").trains():
    mod0 = data["FXE_DET_LPD1M-1/DET/0CH0:xtdf"]["image.data"]
```

These are not the only ways to get data: *Reading data files* describes various other options. karabo_data also has classes to work with detector geometry, described in *AGIPD, LPD & DSSC Geometry*.

CHAPTER

THREE

DOCUMENTATION CONTENTS

3.1 Reading data files

3.1.1 Opening files

You will normally access data from a run, which is stored as a directory containing HDF5 files. You can open a run using `RunDirectory()` with the path of the directory, or using `open_run()` with the proposal number and run number to look up the standard data paths on the Maxwell cluster.

karabo_data.**RunDirectory**(*path*, *include='*'*)

Open data files from a 'run' at European XFEL.

```
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0001")
```

A 'run' is a directory containing a number of HDF5 files with data from the same time period.

Returns a *DataCollection* object.

Parameters

- **path** (*str*) – Path to the run directory containing HDF5 files.
- **include** (*str*) – Wildcard string to filter data files.

karabo_data.**open_run**(*proposal*, *run*, *data='raw'*, *include='*'*)

Access EuXFEL data on the Maxwell cluster by proposal and run number.

```
run = open_run(proposal=700000, run=1)
```

Returns a *DataCollection* object.

Parameters

- **proposal** (*str, int*) – A proposal number, such as 2012, '2012', 'p002012', or a path such as '/gpfs/exfel/exp/SPB/201701/p002012'.
- **run** (*str, int*) – A run number such as 243, '243' or 'r0243'.
- **data** (*str*) – 'raw' or 'proc' (processed) to access data from one of those folders. The default is 'raw'.
- **include** (*str*) – Wildcard string to filter data files.

New in version 0.5.

You can also open a single file. The methods described below all work for either a run or a single file.

karabo_data.**H5File**(*path*)
Open a single HDF5 file generated at European XFEL.

```
file = H5File("RAW-R0017-DA01-S00000.h5")
```

Returns a *DataCollection* object.

Parameters **path** (*str*) – Path to the HDF5 file

3.1.2 Data structure

A run (or file) contains data from various *sources*, each of which has *keys*. For instance, `SA1_XTD2_XGM/XGM/DOOCS` is one source, for an 'XGM' device which monitors the beam, and its keys include `beamPosition.ixPos` and `beamPosition.iyPos`.

European XFEL produces ten *pulse trains* per second, each of which can contain up to 2700 X-ray pulses. Each pulse train has a unique train ID, which is used to refer to all data associated with that 0.1 second window.

class karabo_data.**DataCollection**

train_ids
A list of the train IDs included in this data. The data recorded may not be the same for each train.

control_sources
A set of the control source names in this data, in the format `"SA3_XTD10_VAC/TSENS/S30100K"`. Control data is always recorded exactly once per train.

instrument_sources
A set of the instrument source names in this data, in the format `"FXE_DET_LPD1M-1/DET/15CH0:xtdf"`. Instrument data may be recorded zero to many times per train.

all_sources
A set of names for both instrument and control sources. This is the union of the two sets above.

keys_for_source(*source*)
Get a set of key names for the given source

If you have used *select()* to filter keys, only selected keys are returned.

Only one file is used to find the keys. Within a run, all files should have the same keys for a given source, but if you use *union()* to combine two runs where the source was configured differently, the result can be unpredictable.

get_data_counts(*source*, *key*)
Get a count of data points in each train for the given data field.

Returns a pandas series with an index of train IDs.

Parameters

- **source** (*str*) – Source name, e.g. "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "image.data".

info()
Show information about the selected data.

3.1.3 Getting data by source & key

Where data will fit into memory, it's usually quickest and most convenient to load it like this.

class karabo_data.**DataCollection**

get_array(*source, key, extra_dims=None, roi=by_index[...]*)
Return a labelled array for a particular data field.

```
arr = run.get_array("SA3_XTD10_PES/ADC/1:network", "digitizers.channel_4_A.
↪raw.samples")
```

This should work for any data. The first axis of the returned data will be labelled with the train IDs.

Parameters

- **source** (*str*) – Device name with optional output channel, e.g. "SA1_XTD2_XGM/DOOCS/MAIN" or "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "beamPosition.iyPos.value" or "header.linkId".
- **extra_dims** (*list of str*) – Name extra dimensions in the array. The first dimension is automatically called 'train'. The default for extra dimensions is dim_0, dim_1, ...
- **roi** (*by_index*) – The region of interest. This expression selects data in all dimensions apart from the first (trains) dimension. If the data holds a 1D array for each entry, roi=by_index[:8] would get the first 8 values from every train. If the data is 2D or more at each entry, selection looks like roi=by_index[:8, 5:10] .

See also:

xarray documentation How to use the arrays returned by *get_array()*

Working with non-detector data Examples using xarray & pandas with EuXFEL data

get_dask_array(*source*, *key*)
Get a Dask array for the specified data field.

Dask is a system for lazy parallel computation. This method doesn't actually load the data, but gives you an array-like object which you can operate on. Dask loads the data and calculates results when you ask it to, e.g. by calling a `.compute()` method. See the Dask documentation for more details.

If your computation depends on reading lots of data, consider creating a dask.distributed.Client before calling this. If you don't do this, Dask uses threads by default, which is not efficient for reading HDF5 files.

Parameters

- **source** (*str*) – Source name, e.g. "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "image.data".

See also:

Dask Array documentation How to use the objects returned by *get_dask_array()*

Averaging detector data with Dask An example using Dask with EuXFEL data

get_series(*source*, *key*)
Return a pandas Series for a particular data field.

```
s = run.get_series("SA1_XTD2_XGM/XGM/DOOCS", "beamPosition.ixPos")
```

This only works for 1-dimensional data.

Parameters

- **source** (*str*) – Device name with optional output channel, e.g. "SA1_XTD2_XGM/DOOCS/MAIN" or "SPB_DET_AGIPD1M-1/DET/7CH0:xtdf"
- **key** (*str*) – Key of parameter within that device, e.g. "beamPosition.iyPos.value" or "header.linkId". The data must be 1D in the file.

get_dataframe(*fields=None*, ***, *timestamps=False*)
Return a pandas dataframe for given data fields.

```
df = run.get_dataframe(fields=[
    ("*_XGM/*", "*.i[xy]Pos"),
    ("*_XGM/*", "*.photonFlux")
])
```

This links together multiple 1-dimensional datasets as columns in a table.

Parameters

- **fields** (*dict or list, optional*) – Select data sources and keys to include in the dataframe. Selections are defined by lists or dicts as in *select()*.
- **timestamps** (*bool*) – If false (the default), exclude the timestamps associated with each control data field.

See also:

pandas documentation How to use the objects returned by *get_series()* and *get_dataframe()*

Working with non-detector data Examples using xarray & pandas with EuXFEL data

get_virtual_dataset(*source*, *key*, *filename=None*)
Create an HDF5 virtual dataset for a given source & key

A dataset looks like a multidimensional array, but the data is loaded on-demand when you access it. So it's suitable as an interface to data which is too big to load entirely into memory.

This returns an h5py.Dataset object. This exists in a real file as a 'virtual dataset', a collection of links pointing to the data in real datasets. If *filename* is passed, the file is written at that path, overwriting if it already exists. Otherwise, it uses a new temp file.

To access the dataset from other worker processes, give them the name of the created file along with the path to the dataset inside it (accessible as `ds.name`). They will need at least HDF5 1.10 to access the virtual dataset, and they must be on a system with access to the original data files, as the virtual dataset points to those.

New in version 0.5.

See also:

Parallel processing with a virtual dataset

3.1.4 Getting data by train

Some kinds of data, e.g. from AGIPD, are too big to load a whole run into memory at once. In these cases, it's convenient to load one train at a time.

When accessing data like this, it's worth selecting which sources you're interested in, either using *select()*, or the devices= parameter. This avoids reading all the other data.

class karabo_data.**DataCollection**

trains(*devices=None*, *train_range=None*, ***, *require_all=False*)
Iterate over all trains in the data and gather all sources.

```
run = Run('/path/to/my/run/r0123')
for train_id, data in run.select("*/DET/*", "image.data").trains():
    mod0 = data["FXE_DET_LPD1M-1/DET/0CH0:xtdf"]["image.data"]
```

Parameters

- **devices** (*dict or list, optional*) – Filter data by sources and keys. Refer to *select()* for how to use this.
- **train_range** (*by_id or by_index object, optional*) – Iterate over only selected trains, by train ID or by index. Refer to *select_trains()* for how to use this.
- **require_all** (*bool*) – False (default) returns any data available for the requested trains. True skips trains which don't have all the selected data; this only makes sense if you make a selection with *devices* or *select()*.

Yields

- **tid** (*int*) – The train ID of the returned train
- **data** (*dict*) – The data for this train, keyed by device name

train_from_id(*train_id*, *devices=None*)
Get train data for specified train ID.

Parameters

- **train_id** (*int*) – The train ID
- **devices** (*dict or list, optional*) – Filter data by sources and keys. Refer to *select()* for how to use this.

Returns

- **tid** (*int*) – The train ID of the returned train
- **data** (*dict*) – The data for this train, keyed by device name

Raises **KeyError** – if *train_id* is not found in the run.

train_from_index(*train_index*, *devices=None*)
Get train data of the nth train in this data.

Parameters

- **train_index** (*int*) – Index of the train in the file.
- **devices** (*dict or list, optional*) – Filter data by sources and keys. Refer to *select()* for how to use this.

Returns

- **tid** (*int*) – The train ID of the returned train
- **data** (*dict*) – The data for this train, keyed by device name

3.1.5 Selecting & combining data

These methods all return a new *DataCollection* object with the selected data, so you use them like this:

```
sel = run.select("*/XGM/*")
# sel includes only XGM sources
# run still includes all the data
```

class karabo_data.**DataCollection**

select (*seln_or_source_glob*, *key_glob='*'*)

Select a subset of sources and keys from this data.

There are three possible ways to select data:

1. With two glob patterns (see below) for source and key names:

   ```
   # Select data in the image group for any detector sources
   sel = run.select('*/DET/*, 'image.*')
   ```

2. With a list of (source, key) glob patterns:

   ```
   # Select image.data and image.mask for any detector sources
   sel = run.select([('*/DET/*, 'image.data'), ('*/DET/*, 'image.mask')])
   ```

   Data is included if it matches any of the pattern pairs.

3. With a dict of source names mapped to sets of key names (or empty sets to get all keys):

   ```
   # Select image.data from one detector source, and all data from one XGM
   sel = run.select({'SPB_DET_AGIPD1M-1/DET/0CH0:xtdf': {'image.data'},
                     'SA1_XTD2_XGM/XGM/DOOCS': set()})
   ```

   Unlike the others, this option *doesn't* allow glob patterns. It's a more precise but less convenient option for code that knows exactly what sources and keys it needs.

Returns a new *DataCollection* object for the selected data.

Note: 'Glob' patterns may be familiar from selecting files in a Unix shell. `*` matches anything, so `*/DET/*` selects sources with "/DET/" anywhere in the name. There are several kinds of wildcard:

- `*`: anything
- `?`: any single character
- `[xyz]`: one character, "x", "y" or "z"
- `[0-9]`: one digit character
- `[!xyz]`: one character, *not* x, y or z

Anything else in the pattern must match exactly. It's case-sensitive, so "x" does not match "X".

---

deselect (*seln_or_source_glob*, *key_glob='*'*)
Select everything except the specified sources and keys.

This takes the same arguments as *select()*, but the sources and keys you specify are dropped from the selection.

Returns a new *DataCollection* object for the remaining data.

select_trains (*train_range*)
Select a subset of trains from this data.

Choose a slice of trains by train ID:

```
from karabo_data import by_id
sel = run.select_trains(by_id[142844490:142844495])
```

Or select a list of trains:

```
sel = run.select_trains(by_id[[142844490, 142844493, 142844494]])
```

Or select trains by index within this collection:

```
from karabo_data import by_index
sel = run.select_trains(by_index[:5])
```

Returns a new *DataCollection* object for the selected trains.

Raises **ValueError** – If given train IDs do not overlap with the trains in this data.

union (**others*)
Join the data in this collection with one or more others.

This can be used to join multiple sources for the same trains, or to extend the same sources with data for further trains. The order of the datasets doesn't matter.

Returns a new *DataCollection* object.

3.1.6 Writing selected data

class karabo_data.**DataCollection**

write (*filename*)
Write the selected data to a new HDF5 file

You can choose a subset of the data using methods like *select()* and *select_trains()*, then use this write it to a new, smaller file.

The target filename will be overwritten if it already exists.

write_virtual (*filename*)
Write an HDF5 file with virtual datasets for the selected data.

This doesn't copy the data, but each virtual dataset provides a view of data spanning multiple sequence files, which can be accessed as if it had been copied into one big file.

This is *not* the same as building virtual datasets to combine multi-module detector data. See *AGIPD, LPD & DSSC data* for that.

Creating and reading virtual datasets requires HDF5 version 1.10.

The target filename will be overwritten if it already exists.

3.1.7 Missing data

What happens if some data was not recorded for a given train?

Control data is duplicated for each train until it changes. If the device cannot send changes, the last values will be recorded for each subsequent train until it sends changes again. There is no general way to distinguish this scenario from values which genuinely aren't changing.

Parts of instrument data may be missing from the file. These will also be missing from the data returned by `karabo_data`:

- The train-oriented methods *`trains()`*, *`train_from_id()`*, and *`train_from_index()`* give you dictionaries keyed by source and key name. Sources and keys are only included if they have data for that train.
- *`get_array()`*, and *`get_series()`* skip over trains which are missing data. The indexes on the returned DataArray or Series objects link the returned data to train IDs. Further operations with xarray or pandas may drop misaligned data or introduce fill values.
- *`get_dataframe()`* includes rows for which any column has data. Where some but not all columns have data, the missing values are filled with `NaN` by pandas' missing data handling.

Missing data does not necessarily mean that something has gone wrong: some devices send data at less than 10 Hz (the train rate), so they always have gaps between updates.

3.1.8 Data problems

If you encounter problems accessing data with `karabo_data`, there may be problems with the data files themselves. Use the `karabo-data-validate` command to check for this (see *Checking data files*).

Here are some problems we've seen, and possible solutions or workarounds:

- Indexes point to data beyond the end of datasets: this has previously been caused by bugs in the detector calibration pipeline. If you see this in calibrated data (in the `proc/` folder), ask for the relevant runs to be re-calibrated.
- Train IDs are not strictly increasing: issues with the timing system when the data is recorded can create an occasional train ID which is completely out of sequence. Usually it seems to be possible to ignore this and use the remaining data, but if you have any issues, please let us know.
  - In one case, a train ID had the maximum possible value (2^{64} - 1), causing *`info()`* to fail. You can select everything except this train using *`select_trains()`*:

```
from karabo_data import by_id
sel = run.select_trains(by_id[:2**64-1])
```

If you're having problems with karabo_data, you can also try searching previously reported issues to see if anyone has encountered similar symptoms.

3.1.9 Cached run data maps

When you open a run in karabo_data, it needs to know what data is in each file. Each file has metadata describing its contents, but reading this from every file is slow, especially on GPFS. karabo_data therefore tries to cache this information the first time a run is opened, and reuse it when opening that run again.

This should happen automatically, without the user needing to know about it. You only need these details if you think caching may be causing problems.

- Caching is triggered when you use *RunDirectory()* or *open_run()*.
- There are two possible locations for the cached data map:
  - In the run directory: `(run dir)/karabo_data_map.json`.
  - In the proposal scratch directory: `(proposal dir)/scratch/.karabo_data_maps/raw_r0032.json`. This will normally be the one used on Maxwell, as users can't write to the run directory.
- The format is a JSON array, with an object for each file in the run.
  - This holds the list of train IDs in the file, and the lists of control and instrument sources.
  - It also stores the file size and last modified time of each data file, to check if the file has changed since the cache was created. If either of these attributes doesn't match, karabo_data ignores the cached information and reads the metadata from the HDF5 file.
- If any file in the run wasn't listed in the data map, or its entry was outdated, a new data map is written automatically. It tries the same two locations described above, but it will continue without error if it can't write to either.

JSON was chosen as it can be easily inspected manually, and it's reasonably efficient to load the entire file.

3.2 AGIPD, LPD & DSSC data

These data from AGIPD, LPD and DSSC is spread out in separate files. `karabo_data` includes convenient interfaces to access this data, pulling together the separate modules into a single array.

class karabo_data.components.**AGIPD1M**(*data: karabo_data.reader.DataCollection*, *detector_name=None*, *modules=None*, ***, *min_modules=1*)

An interface to AGIPD-1M data.

Parameters

- **data** (DataCollection) – A data collection, e.g. from RunDirectory.
- **modules** (*set of ints, optional*) – Detector module numbers to use. By default, all available modules are used.
- **detector_name** (*str, optional*) – Name of a detector, e.g. 'SPB_DET_AGIPD1M-1'. This is only needed if the dataset includes more than one AGIPD detector.
- **min_modules** (*int*) – Include trains where at least n modules have data. Default is 1.

The methods of this class are identical to those of *LPD1M*, below.

class karabo_data.components.**DSSC1M**(*data: karabo_data.reader.DataCollection*, *detector_name=None*, *modules=None*, ***, *min_modules=1*)

An interface to DSSC-1M data.

Parameters

- **data** (DataCollection) – A data collection, e.g. from RunDirectory.
- **modules** (*set of ints, optional*) – Detector module numbers to use. By default, all available modules are used.
- **detector_name** (*str, optional*) – Name of a detector, e.g. 'SCS_DET_DSSC1M-1'. This is only needed if the dataset includes more than one DSSC detector.
- **min_modules** (*int*) – Include trains where at least n modules have data. Default is 1.

The methods of this class are identical to those of *LPD1M*, below.

class karabo_data.components.**LPD1M**(*data: karabo_data.reader.DataCollection*, *detector_name=None*, *modules=None*, ***, *min_modules=1*)

An interface to LPD-1M data.

Parameters

- **data** (DataCollection) – A data collection, e.g. from RunDirectory.
- **modules** (*set of ints, optional*) – Detector module numbers to use. By default, all available modules are used.
- **detector_name** (*str, optional*) – Name of a detector, e.g. 'FXE_DET_LPD1M-1'. This is only needed if the dataset includes more than one LPD detector.
- **min_modules** (*int*) – Include trains where at least n modules have data. Default is 1.

get_array(*key, pulses=by_index[:]*)

Get a labelled array of detector data

Parameters

- **key** (*str*) – The data to get, e.g. 'image.data' for pixel values.
- **pulses** (*by_id or by_index*) – Select the pulses to include from each train. by_id selects by pulse ID, by_index by index within the data being read. The default includes all pulses. Only used for per-train data.

trains(*pulses=by_index[:]*)

Iterate over trains for detector data.

Parameters **pulses** (*by_index or by_id*) – Select which pulses to include for each train. The default is to include all pulses.

Yields **train_data** (*dict*) – A dictionary mapping key names (e.g. `image.data`) to labelled arrays.

write_virtual_cxi(*filename*)

Write a virtual CXI file to access the detector data.

The virtual datasets in the file provide a view of the detector data as if it was a single huge array, but without copying the data. Creating and using virtual datasets requires HDF5 1.10.

Parameters **filename** (*str*) – The file to be written. Will be overwritten if it already exists.

See also:

Accessing LPD data: An example using the class above.

If you get data for a train from the main `DataCollection` interface, there is also another way to combine detector modules from AGIPD or LPD:

karabo_data.**stack_detector_data**(*train*, *data*, *axis=-3*, *modules=16*, *fillvalue=nan*, *real_array=True*)

Stack data from detector modules in a train.

Parameters

- **train** (*dict*) – Train data.
- **data** (*str*) – The path to the device parameter of the data you want to stack, e.g. 'image.data'.
- **axis** (*int*) – Array axis on which you wish to stack (default is -3).
- **modules** (*int*) – Number of modules composing a detector (default is 16).
- **fillvalue** (*number*) – Value to use in place of data for missing modules. The default is nan (not a number) for floating-point data, and 0 for integers.
- **real_array** (*bool*) – If True (default), copy the data together into a real numpy array. If False, avoid copying the data and return a limited array-like wrapper around the existing arrays. This is sufficient for assembling images using detector geometry, and allows better performance.

Returns **combined** – Stacked data for requested data path.

Return type numpy.array

3.3 Streaming data over ZeroMQ

Karabo Bridge provides access to live data during the experiment over a ZeroMQ socket. The `karabo_data` Python package can stream data from files using the same protocol. You can use this to test code which expects to receive data from Karabo Bridge, or use the same code for analysing live data and stored data.

To stream the data from a file or run unmodified, use the command:

```
karabo-bridge-serve-files /gpfs/exfel/exp/SPB/201830/p900022/raw/r0034 4545
```

The number (4545) must be an unused TCP port above 1024. It will bind to this and stream the data to any connected clients.

We provide Karabo bridge clients as Python and C++ libraries.

If you want to do some processing on the data before streaming it, you can use this Python interface to send it out:

class karabo_data.export.**ZMQStreamer**(*port*, *maxlen=10*, *protocol_version='2.2'*, *dummy_timestamps=False*)

ZeroMQ inteface sending data over a TCP socket.

```
# Server:
serve = ZMQStreamer(1234)
serve.start()

for tid, data in run.trains():
    result = important_processing(data)
    serve.feed(result)

# Client:
from karabo_bridge import Client
client = Client('tcp://server.hostname:1234')
data = client.next()
```

Parameters

- **port** (*int*) – Local TCP port to bind socket to

- **maxlen** (*int, optional*) – How many trains to cache before sending (default: 10)
- **protocol_version** (*('1.0' | '2.1')*) – Which version of the bridge protocol to use. Defaults to the latest version implemented.
- **dummy_timestamps** (*bool*) – Some tools (such as OnDA) expect the timestamp information to be in the messages. We can't give accurate timestamps where these are not in the file, so this option generates fake timestamps from the time the data is fed in.

start()
Start a zmq.REP socket.

feed(*data*, *metadata=None*)
Push data to the sending queue.

This blocks if the queue already has *maxlen* items waiting to be sent.

Parameters

- **data** (*dict*) – Contains train data. The dictionary has to follow the karabo_bridge protocol structure:
  - keys are source names
  - values are dict, where the keys are the parameter names and values must be python built-in types or numpy.ndarray.
- **metadata** (*dict, optional*) – Contains train metadata. The dictionary has to follow the karabo_bridge protocol structure:
  - keys are (str) source names
  - values (dict) should contain the following items:
    * 'timestamp' Unix time with subsecond resolution
    * 'timestamp.sec' Unix time with second resolution
    * 'timestamp.frac' fractional part with attosecond resolution
    * 'timestamp.tid' is European XFEL train unique ID

```
{
    'source': 'sourceName'  # str
    'timestamp': 1234.567890  # float
    'timestamp.sec': '1234'  # str
    'timestamp.frac': '567890000000000000'  # str
    'timestamp.tid': 1234567890  # int
}
```

If the metadata dict is not provided it will be extracted from 'data' or an empty dict if 'metadata' key is missing from a data source.

3.4 Checking data files

karabo_data includes a tool to check the integrity of data files. You can pass it a run:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0803
```

Or a single data file:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0803/RAW-R0803-AGIPD00-
↪S00000.h5
```

The checks are informed by problems we have encountered with data files in the past. Currently, it checks that:

- All `.h5` files in a run can be opened, and the run contains at least one usable file.
- The list of train IDs in a file has no zeros except for padding at the end.
- Each train ID in a file is greater than the one before it.
- The indexes have the same number of entries as train IDs.
- The indexes do not point to data beyond the end of a dataset.
- The indexes point to the start of the dataset, and then to successive chunks for successive trains, without gaps or overlaps between them.

If any checks fail, the output will contain details, and the exit code will be non-zero. An exit code of 0 means that the checks all passed. This is the standard convention for command line tools to indicate success or failure.

3.5 AGIPD, LPD & DSSC Geometry

The AGIPD and LPD detectors are made up of several sensor modules, from which separate streams of data are recorded. Inspecting or processing data from these detectors therefore depends on knowing how the modules are arranged. The module *karabo_data.geometry2* handles this information.

All the coordinates used in this module are from the detector centre. This should be roughly where the beam passes through the detector. They follow the standard European XFEL axis orientations, with x increasing to the left (looking along the beam), and y increasing upwards.

Note: This module includes methods to assemble data into a single array. This is sufficient for a quick examination of detector images, but the detector pixels may not line up with the grid imposed by a single array. For accurate analysis, it's best to use a tool that can process geometry internally with sub-pixel precision.

3.5.1 AGIPD-1M

AGIPD-1M consists of 16 modules of 512×128 pixels each. Each module is further subdivided into 8 tiles. The layout of tiles within a module is fixed by the manufacturing process, but this geometry code works with a position for each tile.

class karabo_data.geometry2.**AGIPD_1MGeometry**(*modules*, *filename='No file'*)
Detector layout for AGIPD-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

You won't normally instantiate this class directly: use one of the constructor class methods to create or load a geometry.

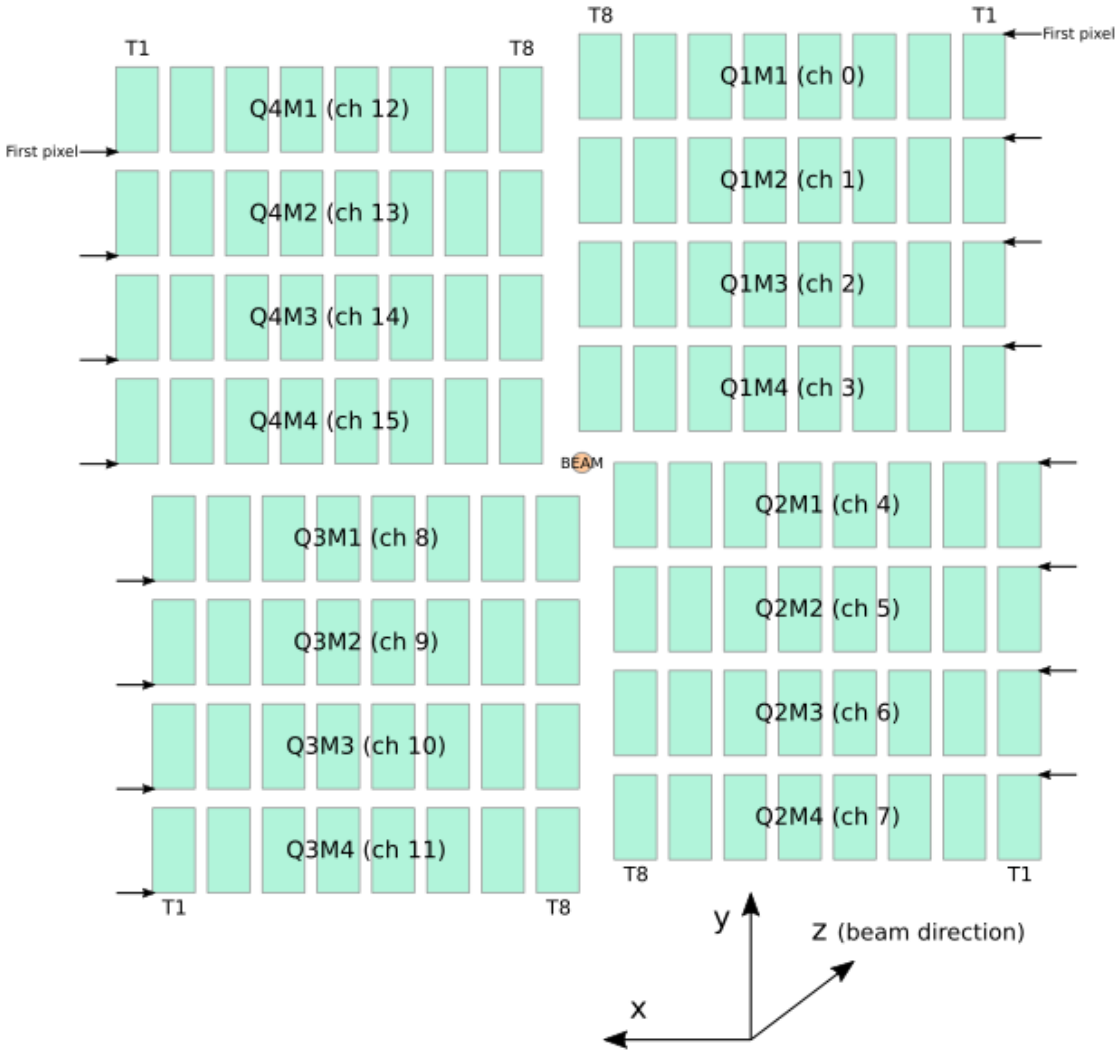


Fig. 1: The approximate layout of AGIPD-1M, in a front view (looking along the beam).

classmethod from_quad_positions(*quad_pos*, *asic_gap=2*, *panel_gap=29*, *unit=0.0002*)
Generate an AGIPD-1M geometry from quadrant positions.

This produces an idealised geometry, assuming all modules are perfectly flat, aligned and equally spaced within their quadrant.

The quadrant positions are given in pixel units, referring to the first pixel of the first module in each quadrant, corresponding to data channels 0, 4, 8 and 12.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

To give positions in units other than pixels, pass the *unit* parameter as the length of the unit in metres. E.g. `unit=1e-3` means the coordinates are in millimetres.

classmethod from_crystfel_geom(*filename*)
Read a CrystFEL format (.geom) geometry file.

Returns a new geometry object.

write_crystfel_geom(*filename*, ***, *data_path='/entry_1/instrument_1/detector_1/data'*, *mask_path=None*, *dims=('frame'*, *'modno'*, *'ss'*, *'fs')*, *adu_per_ev=None*, *clen=None*, *photon_energy=None*)
Write this geometry to a CrystFEL format (.geom) geometry file.

Parameters

- **filename** (*str*) – Filename of the geometry file to write.
- **data_path** (*str*) – Path to the group that contains the data array in the hdf5 file. Default: `'/entry_1/instrument_1/detector_1/data'`.
- **mask_path** (*str*) – Path to the group that contains the mask array in the hdf5 file.
- **dims** (*tuple*) – Dimensions of the data. Extra dimensions, except for the defaults, should be added by their index, e.g. ('frame', 'modno', 0, 'ss', 'fs') for raw data. Default: `('frame', 'modno', 'ss', 'fs')`. Note: the dimensions must contain frame, ss, fs.
- **adu_per_ev** (*float*) – ADU (analog digital units) per electron volt for the considered detector.
- **clen** (*float*) – Distance between sample and detector in meters
- **photon_energy** (*float*) – Beam wave length in eV

get_pixel_positions(*centre=True*)
Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

to_distortion_array(*allow_negative_xy=False*)
Return distortion matrix for AGIPD detector, suitable for pyFAI.

Parameters **allow_negative_xy** (*bool*) – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

Returns

out – Array of float 32 with shape (8192, 128, 4, 3). The dimensions mean:

- 8192 = 16 modules * 512 pixels (slow scan axis)

- 128 pixels (fast scan axis)
- 4 corners of each pixel
- 3 numbers for z, y, x

Return type ndarray

plot_data_fast(*data*, ***, *axis_units='px'*, *frontview=True*, *ax=None*, *figsize=None*, *colorbar=True*, ***kwargs*)

Plot data from the detector using this geometry.

This approximates the geometry to align all pixels to a 2D grid.

Returns a matplotlib axes object.

Parameters

- **data** (*ndarray*) – Should have exactly 3 dimensions, for the modules, then the slow scan and fast scan pixel dimensions.
- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.
- **ax** (*~matplotlib.axes.Axes* object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize** (*tuple*) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar** (*bool, dict*) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to *~matplotlib.imshow*

position_modules_fast(*data*, *out=None*)

Assemble data from this detector according to where the pixels are.

This approximates the geometry to align all pixels to a 2D grid.

Parameters

- **data** (*ndarray*) – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.
- **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use *output_array_for_position_fast()* to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

Returns

- **out** (*ndarray*) – Array with one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
- **centre** (*ndarray*) – (y, x) pixel location of the detector centre in this geometry.

output_array_for_position_fast(*extra_shape=()*, *dtype=<class 'numpy.float32'>*)

Make an empty output array to use with position_modules_fast

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to *position_modules_fast()*. By default, it allocates a new array on each call, which can be slow.

Parameters

- **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.
- **dtype** (*optional (Default: np.float32)*) –

position_modules_interpolate(*data*)

Assemble data from this detector according to where the pixels are.

This performs interpolation, which is very slow. Use *position_modules_fast()* to get a pixel-aligned approximation of the geometry.

Parameters **data** (*ndarray*) – The three dimensions should be channelno, pixel_ss, pixel_fs (lengths 16, 512, 128). ss/fs are slow-scan and fast-scan.

Returns

- **out** (*ndarray*) – Array with the one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
- **centre** (*ndarray*) – (y, x) pixel location of the detector centre in this geometry.

inspect(*axis_units='px'*, *frontview=True*)

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

Parameters

- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.

compare(*other*, *scale=1.0*)

Show a comparison of this geometry with another in a 2D plot.

This shows the current geometry like *inspect()*, with the addition of arrows showing how each panel is shifted in the other geometry.

Parameters

- **other** (AGIPD_1MGeometry) – A second geometry object to compare with this one.
- **scale** (*float*) – Scale the arrows showing the difference in positions. This is useful to show small differences clearly.

data_coords_to_positions(*module_no*, *slow_scan*, *fast_scan*)

Convert data array coordinates to physical positions

Data array coordinates are how you might refer to a pixel in an array of detector data: module number, and indices in the slow-scan and fast-scan directions. But coordinates in the two pixel dimensions aren't necessarily integers, e.g. if they refer to the centre of a peak.

module_no, fast_scan and slow_scan should all be numpy arrays of the same shape. module_no should hold integers, starting from 0, so 0: Q1M1, 1: Q1M2, etc.

slow_scan and fast_scan describe positions within that module. They may hold floats for sub-pixel positions. In both, 0.5 is the centre of the first pixel.

Returns an array of similar shape with an extra dimension of length 3, for (x, y, z) coordinates in metres.

See also:

Detector geometry for AGIPD demonstrates using this method.

3.5.2 LPD-1M

LPD-1M consists of 16 supermodules of 256×256 pixels each. Each supermodule is further subdivided into 16 sensor tiles, which this geometry code can position independently.

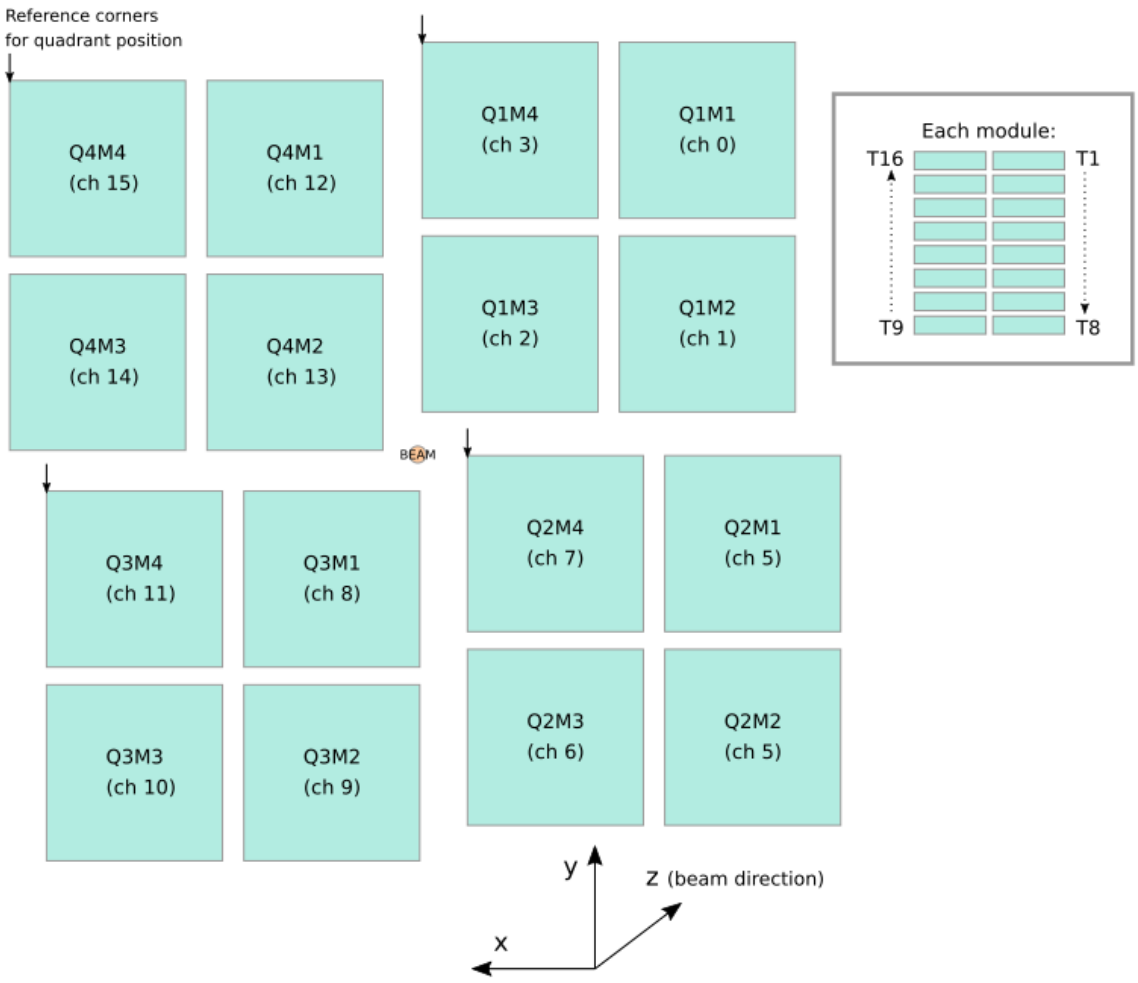


Fig. 2: The approximate layout of LPD-1M, in a front view (looking along the beam).

class karabo_data.geometry2.**LPD_1MGeometry**(*modules*, *filename='No file'*)

Detector layout for LPD-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

You won't normally instantiate this class directly: use one of the constructor class methods to create or load a geometry.

classmethod from_quad_positions(*quad_pos*, *, *unit=0.001*, *asic_gap=None*, *panel_gap=None*)

Generate an LPD-1M geometry from quadrant positions.

This produces an idealised geometry, assuming all modules are perfectly flat, aligned and equally spaced within their quadrant.

The quadrant positions refer to the corner of each quadrant where module 4, tile 16 is positioned. This is the corner of the last pixel as the data is stored. In the initial detector layout, the corner positions are for the top left corner of the quadrant, looking along the beam.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

Parameters

- **quad_pos** (*list of 2-tuples*) – (x, y) coordinates of the last corner (the one by module 4) of each quadrant.
- **unit** (*float, optional*) – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.
- **asic_gap** (*float, optional*) – The gap between adjacent tiles/ASICs. The default is 4 pixels.
- **panel_gap** (*float, optional*) – The gap between adjacent modules/panels. The default is 4 pixels.

classmethod from_h5_file_and_quad_positions(*path*, *positions*, *unit=0.001*)

Load an LPD-1M geometry from an XFEL HDF5 format geometry file

The quadrant positions are not stored in the file, and must be provided separately. By default, both the quadrant positions and the positions in the file are measured in millimetres; the unit parameter controls this.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

This version of the code only handles x and y translation, as this is all that is recorded in the initial LPD geometry file.

Parameters

- **path** (*str*) – Path of an EuXFEL format (HDF5) geometry file for LPD.
- **positions** (*list of 2-tuples*) – (x, y) coordinates of the last corner (the one by module 4) of each quadrant.
- **unit** (*float, optional*) – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.

classmethod from_crystfel_geom(*filename*)

Read a CrystFEL format (.geom) geometry file.

Returns a new geometry object.

write_crystfel_geom(*filename*, *, *data_path='/entry_1/instrument_1/detector_1/data'*, *mask_path=None*, *dims=('frame'*, *'modno'*, *'ss'*, *'fs')*, *adu_per_ev=None*, *clen=None*, *photon_energy=None*)

Write this geometry to a CrystFEL format (.geom) geometry file.

Parameters

- **filename** (*str*) – Filename of the geometry file to write.
- **data_path** (*str*) – Path to the group that contains the data array in the hdf5 file. Default: `'/entry_1/instrument_1/detector_1/data'`.
- **mask_path** (*str*) – Path to the group that contains the mask array in the hdf5 file.
- **dims** (*tuple*) – Dimensions of the data. Extra dimensions, except for the defaults, should be added by their index, e.g. ('frame', 'modno', 0, 'ss', 'fs') for raw data. Default: `('frame', 'modno', 'ss', 'fs')`. Note: the dimensions must contain frame, ss, fs.
- **adu_per_ev** (*float*) – ADU (analog digital units) per electron volt for the considered detector.
- **clen** (*float*) – Distance between sample and detector in meters
- **photon_energy** (*float*) – Beam wave length in eV

get_pixel_positions(*centre=True*)

Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

to_distortion_array(*allow_negative_xy=False*)

Return distortion matrix for LPD detector, suitable for pyFAI.

Parameters **allow_negative_xy** (*bool*) – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

Returns

out – Array of float 32 with shape (4096, 256, 4, 3). The dimensions mean:

- 4096 = 16 modules * 256 pixels (slow scan axis)
- 256 pixels (fast scan axis)
- 4 corners of each pixel
- 3 numbers for z, y, x

Return type ndarray

plot_data_fast(*data*, ***, *axis_units='px'*, *frontview=True*, *ax=None*, *figsize=None*, *colorbar=True*, ***kwargs*)

Plot data from the detector using this geometry.

This approximates the geometry to align all pixels to a 2D grid.

Returns a matplotlib axes object.

Parameters

- **data** (*ndarray*) – Should have exactly 3 dimensions, for the modules, then the slow scan and fast scan pixel dimensions.
- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.

- **ax** (*~matplotlib.axes.Axes* object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize** (*tuple*) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar** (*bool, dict*) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to *~matplotlib.imshow*

position_modules_fast (*data*, *out=None*)
Assemble data from this detector according to where the pixels are.

This approximates the geometry to align all pixels to a 2D grid.

Parameters

- **data** (*ndarray*) – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.
- **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use *output_array_for_position_fast()* to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

Returns

- **out** (*ndarray*) – Array with one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
- **centre** (*ndarray*) – (y, x) pixel location of the detector centre in this geometry.

output_array_for_position_fast (*extra_shape=()*, *dtype=<class 'numpy.float32'>*)
Make an empty output array to use with position_modules_fast

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to *position_modules_fast()*. By default, it allocates a new array on each call, which can be slow.

Parameters

- **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.
- **dtype** (*optional (Default: np.float32)*) –

inspect (*axis_units='px'*, *frontview=True*)
Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

Parameters

- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.

data_coords_to_positions (*module_no*, *slow_scan*, *fast_scan*)
Convert data array coordinates to physical positions

Data array coordinates are how you might refer to a pixel in an array of detector data: module number, and indices in the slow-scan and fast-scan directions. But coordinates in the two pixel dimensions aren't necessarily integers, e.g. if they refer to the centre of a peak.

module_no, fast_scan and slow_scan should all be numpy arrays of the same shape. module_no should hold integers, starting from 0, so 0: Q1M1, 1: Q1M2, etc.

slow_scan and fast_scan describe positions within that module. They may hold floats for sub-pixel positions. In both, 0.5 is the centre of the first pixel.

Returns an array of similar shape with an extra dimension of length 3, for (x, y, z) coordinates in metres.

See also:

Detector geometry for AGIPD demonstrates using this method.

3.5.3 DSSC-1M

DSSC-1M consists of 16 modules of 128×512 pixels each. Each module is further subdivided into 2 sensor tiles, which this geometry code can position independently.

The pixels in each DSSC module are tesselating hexagons. This geometry code does not yet handle this: it treats the pixels as rectangles to simplify processing. This is adequate for previewing detector images, but some pixels will be approximately half a pixel width from their true position.

class karabo_data.geometry2.**DSSC_1MGeometry**(*modules*, *filename='No file'*)

Detector layout for DSSC-1M

The coordinates used in this class are 3D (x, y, z), and represent metres.

You won't normally instantiate this class directly: use one of the constructor class methods to create or load a geometry.

classmethod from_h5_file_and_quad_positions(*path*, *positions*, *unit=0.001*)

Load a DSSC geometry from an XFEL HDF5 format geometry file

The quadrant positions are not stored in the file, and must be provided separately. The position given should refer to the bottom right (looking along the beam) corner of the quadrant.

By default, both the quadrant positions and the positions in the file are measured in millimetres; the unit parameter controls this.

The origin of the coordinates is in the centre of the detector. Coordinates increase upwards and to the left (looking along the beam).

This version of the code only handles x and y translation, as this is all that is recorded in the initial LPD geometry file.

Parameters

- **path** (*str*) – Path of an EuXFEL format (HDF5) geometry file for DSSC.
- **positions** (*list of 2-tuples*) – (x, y) coordinates of the last corner (the one by module 4) of each quadrant.
- **unit** (*float, optional*) – The conversion factor to put the coordinates into metres. The default 1e-3 means the numbers are in millimetres.

get_pixel_positions(*centre=True*)

Get the physical coordinates of each pixel in the detector

The output is an array with shape like the data, with an extra dimension of length 3 to hold (x, y, z) coordinates. Coordinates are in metres.

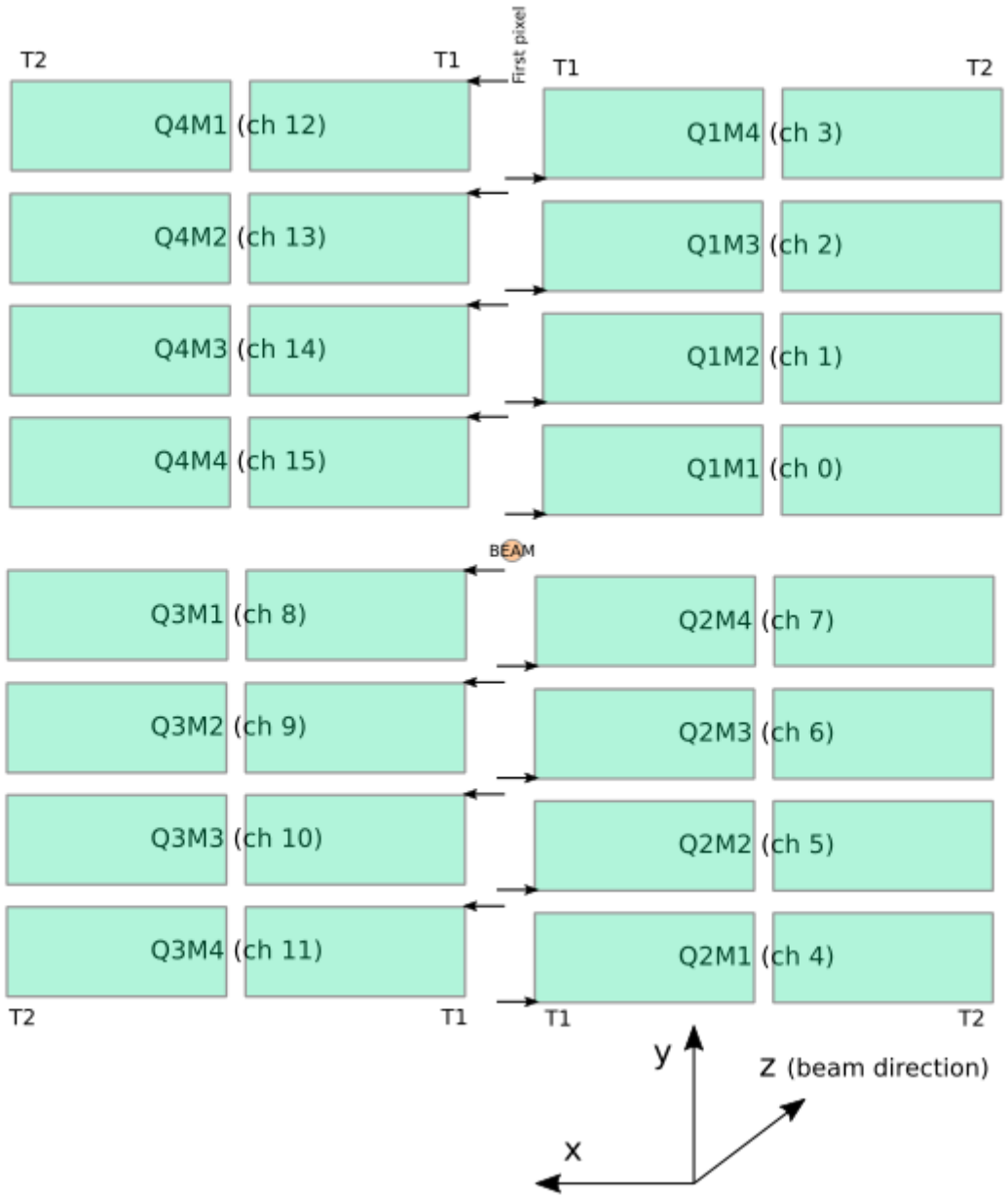


Fig. 3: The approximate layout of DSSC-1M, in a front view (looking along the beam).

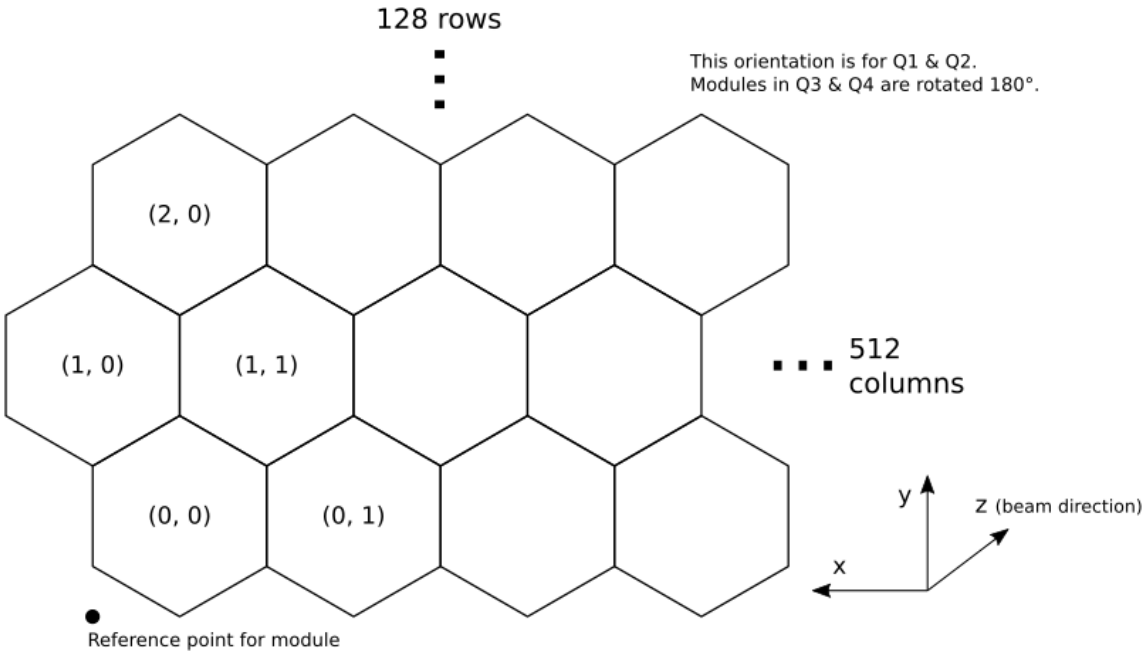


Fig. 4: Detail of hexagonal pixels in the corner of one DSSC module.

If centre=True, the coordinates are calculated for the centre of each pixel. If not, the coordinates are for the first corner of the pixel (the one nearest the [0, 0] corner of the tile in data space).

to_distortion_array(*allow_negative_xy=False*)

Return distortion matrix for DSSC detector, suitable for pyFAI.

Parameters **allow_negative_xy** (*bool*) – If False (default), shift the origin so no x or y coordinates are negative. If True, the origin is the detector centre.

Returns

out – Array of float 32 with shape (2048, 512, 6, 3). The dimensions mean:

- 2048 = 16 modules * 128 pixels (slow scan axis)
- 512 pixels (fast scan axis)
- 6 corners of each pixel
- 3 numbers for z, y, x

Return type ndarray

plot_data_fast(*data*, ***, *axis_units='px'*, *frontview=True*, *ax=None*, *figsize=None*, *colorbar=False*, ***kwargs*)

Plot data from the detector using this geometry.

This approximates the geometry to align all pixels to a 2D grid.

Returns a matplotlib axes object.

Parameters

- **data** (*ndarray*) – Should have exactly 3 dimensions, for the modules, then the slow scan and fast scan pixel dimensions.
- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.
- **ax** (*~matplotlib.axes.Axes* object, optional) – Axes that will be used to draw the image. If None is given (default) a new axes object will be created.
- **figsize** (*tuple*) – Size of the figure (width, height) in inches to be drawn (default: (10, 10))
- **colorbar** (*bool, dict*) – Draw colobar with default values (if boolean is given). Colorbar appearance can be controlled by passing a dictionary of properties.
- **kwargs** – Additional keyword arguments passed to *~matplotlib.imshow*

position_modules_fast (*data*, *out=None*)

Assemble data from this detector according to where the pixels are.

This approximates the geometry to align all pixels to a 2D grid.

Parameters

- **data** (*ndarray*) – The last three dimensions should match the modules, then the slow scan and fast scan pixel dimensions.
- **out** (*ndarray, optional*) – An output array to assemble the image into. By default, a new array is allocated. Use *output_array_for_position_fast()* to create a suitable array. If an array is passed in, it must match the dtype of the data and the shape of the array that would have been allocated. Parts of the array not covered by detector tiles are not overwritten. In general, you can reuse an output array if you are assembling similar pulses or pulse trains with the same geometry.

Returns

- **out** (*ndarray*) – Array with one dimension fewer than the input. The last two dimensions represent pixel y and x in the detector space.
- **centre** (*ndarray*) – (y, x) pixel location of the detector centre in this geometry.

output_array_for_position_fast (*extra_shape=()*, *dtype=<class 'numpy.float32'>*)

Make an empty output array to use with position_modules_fast

You can speed up assembling images by reusing the same output array: call this once, and then pass the array as the `out=` parameter to *position_modules_fast()*. By default, it allocates a new array on each call, which can be slow.

Parameters

- **extra_shape** (*tuple, optional*) – By default, a 2D output array is generated, to assemble a single detector image. If you are assembling multiple pulses at once, pass `extra_shape=(nframes,)` to get a 3D output array.
- **dtype** (*optional (Default: np.float32)*) –

inspect (*axis_units='px'*, *frontview=True*)

Plot the 2D layout of this detector geometry.

Returns a matplotlib Axes object.

Parameters

- **axis_units** (*str*) – Show the detector scale in pixels ('px') or metres ('m').
- **frontview** (*bool*) – If True (the default), x increases to the left, as if you were looking along the beam. False gives a 'looking into the beam' view.

3.6 Command line tools

3.6.1 lsxfel

Examine the contents of an EuXFEL proposal directory, run directory, or HDF5 file:

```
# Proposal directory
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000

# Run directory
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002

# Single file
lsxfel /gpfs/exfel/exp/XMPL/201750/p700000/proc/r0002/CORR-R0034-AGIPD00-S00000.h5
```

3.6.2 karabo-data-validate

Check the structure of an EuXFEL run or HDF5 file:

```
karabo-data-validate /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002
```

If it finds problems with the data, the program will produce a list of them and exit with status 1.

3.6.3 karabo-bridge-serve-files

Stream data from files in the Karabo bridge format. See *Streaming data over ZeroMQ* for more information.

3.6.4 karabo-data-make-virtual-cxi

Make a virtual CXI file to access AGIPD/LPD detector data from a specified run:

```
karabo-data-make-virtual-cxi /gpfs/exfel/exp/XMPL/201750/p700000/proc/r0003 -o xmpl-3.
↪cxi
```

-o <path>, **--output** <path>
The filename to write. Defaults to creating a file in the proposal's scratch directory.

--min-modules <number>
Include trains where at least N modules have data (default 9).

3.7 Data files format

The main unit of data this tool works with is a *run*. A run is data collected in a specific period, and each research proposal given beantime at European XFEL may collect hundreds of runs.

A run is stored as a directory containing HDF5 data files from different sources. These fall into two important categories:

1. Detector data, from the main X-ray detectors in the various experiments.
    - Each detector module writes separate files, e.g. `RAW-R0348-AGIPD00-S00000.h5`. The number in the third part of the filename identifies the module (0 in this example).
    - The detectors in use as of April 2018 are *LPD* and *AGIPD* in the file names. Each has 16 modules numbered 0–15.
2. All the other data, such as motor positions, beam measurements, etc., are recorded through a *data aggregator*, and stored in a file with the letters *DA* in the name, e.g. `RAW-R0450-DA01-S00000.h5`.

The last part of the file name (e.g. `S00000`) is a sequence number. The data within a run may be broken into a number of sequences. So `RAW-R0450-DA01-S00000.h5` and `RAW-R0450-DA01-S00001.h5` will contain data from the same set of devices, with sequence 1 continuing just after the end of sequence 0. Though all data within a run may be broken into sequences, different data sets do not necessarily break at the same point, so the various 'sequence 0' data files in a run do not have corresponding data.

3.7.1 HDF5 file structure

METADATA

The `METADATA` group in an HDF5 file contains three datasets, each of which is a 1D array of strings:

- `METADATA/dataSourceId` lists data groups in the file. The values are either:
    - `CONTROL/` followed by a Karabo device name, e.g. `CONTROL/SA1_XTD2_XGM/DOOCS/MAIN`.
    - `INSTRUMENT/` followed by a Karabo device name, a colon, the name of the output channel, a slash, and the name of a data group (?), e.g. `INSTRUMENT/SA1_XTD2_XGM/DOOCS/MAIN:output/data`
- `METADATA/deviceId` lists the part of each *dataSourceId* after the first slash.
- `METADATA/root` lists the parts before the first slash, so `concat(root, "/", deviceId) == dataSourceId`.

These three data sets always have the same number of values. They may be padded with empty strings, so empty entries are ignored.

INDEX

`INDEX/trainId` is a 1D array of uint64, listing the pulse trains which the file holds data for. This is crucial, since all other data has to be matched up according to train IDs.

For each entry in `METADATA/deviceId`, the `INDEX` group contains two datasets, both uint64 data with the same length as the train IDs:

- `INDEX/{ deviceId }/count`: for each train ID, how many data samples did this device record. This may be 0 if no data was recorded for this train.
- `INDEX/{ deviceId }/first`: for each train ID, the index at which the corresponding data starts in the arrays for this device.

Thus, to find the data for a given train ID, we could do:

```
train_index = trainIds.index(train_id)
first = device_firsts[train_index]
count = device_counts[train_index]
train_data = data[first : first+count]
```

Control data is always (?) recorded once per train, so *count* is 1 and *first* counts up from 0 to the number of trains. Instrument data is more variable.

Some older files use a different index format with first/last/status instead of first/count. In this case, a status of 0 means that no data was recorded for that train.

CONTROL and RUN

For each *CONTROL* entry in `METADATA/dataSourceId`, there is a group with that name in the file. This may have further arbitrarily nested subgroups representing different properties of that device, e.g. `/CONTROL/SA1_XTD2_XGM/DOOCS/MAIN/current/bottom/output`.

The leaves of this tree are pairs of datasets called `timestamp` and `value`. Each dataset has one entry per train, and the `timestamp` record when the value was updated, which is typically less than once per train. The `value` dataset may have extra dimensions, but in most cases it is 1D.

(Does timestamp update if value is re-read but doesn't change?)

`RUN` holds a complete duplicate of the `CONTROL` hierarchy, but each pair of `timestamp` and `value` contain only one entry, taken at the start of the run. There is still a dimension for this, so 2D value datasets in CONTROL have corresponding 2D datasets in RUN, but the first dimension has length 1.

(Is RUN exactly duplicated in subsequent sequence files?)

INSTRUMENT

For each *INSTRUMENT* entry in `METADATA/dataSourceId`, there is a group with that name in the file. Each such group holds a 1D `trainId` dataset, and a number of other datasets (possibly nested in subgroups). All these datasets have the same length in the first dimension: this represents the successive readings taken. The slices defined by the corresponding datasets in *INDEX* work on this dimension.

The `trainId` dataset for each instrument group thus appears to be redundant with the information in INDEX.

3.8 Performance notes

These are some notes on how to load and process data efficiently.

3.8.1 Load data into memory

Where the data you need can fit into memory, it's more efficient to load it in one go using *get_array()*, *get_series()* or *get_dataframe()*, and then work with it using xarray, numpy or pandas. *Working with non-detector data* has some examples of this. The methods to get data by trains—*trains()*, *train_from_id()* and *train_from_index()*—only load the data for one train at once, which saves memory for big data but is slower to process.

Machines in the Maxwell cluster have hundreds of gigabytes of RAM, so it's practical to load many kinds of data completely into memory. However, data for a full run from megahertz detectors such as AGIPD, LPD or DSSC can easily be too much.

The command `free -h` will show the amount of memory on any Linux machine.

3.8.2 Select sources before getting trains

If you do need to use *trains()*, *train_from_id()* or *train_from_index()* to get data for one train at a time, first pick the sources and keys you need with *select()*. Otherwise, you will load the data for every source in the run, which could be very slow.

```
run = RunDirectory("/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0004")

# SLOW: Don't do this!
for tid, train_data in run.trains():
    ...

# Better option: select image data from all detector modules first.
for tid, train_data in run.select('*/DET/*', 'image.data').trains():
    ...
```

The `devices=` parameter for all three train methods does the same thing as using *select()* like this.

3.8.3 Reduce before assembling

Assembling detector images (see *AGIPD, LPD & DSSC Geometry*) is relatively slow. If your analysis involves a reduction step like summing or averaging over a number of images, try to do this on the data from separate modules before assembling them into images.

This also applies more generally: if a step in your processing makes the data smaller, you want to do that step as near the start as possible.

3.9 Reading data with `karabo_data`

This command creates the sample data files used in the rest of this example. These files contain no real data, but they have the same structure as European XFEL's HDF5 data files.

[1]:
```
!python3 -m karabo_data.tests.make_examples
```

```
Written examples.
```

3.9.1 Single files

[2]:
```
!h5ls fxe_control_example.h5
```

```
CONTROL                  Group
INDEX                    Group
INSTRUMENT               Group
METADATA                 Group
RUN                      Group
```

[3]:
```
from karabo_data import H5File
f = H5File('fxe_control_example.h5')
```

[4]:
```
f.control_sources
```

[4]:
```
frozenset({'FXE_XAD_GEC/CAM/CAMERA',
           'SA1_XTD2_XGM/DOOCS/MAIN',
           'SPB_XTD9_XGM/DOOCS/MAIN'})
```

[5]:
```
f.instrument_sources
```

[5]:
```
frozenset({'FXE_XAD_GEC/CAM/CAMERA:daqOutput',
           'SA1_XTD2_XGM/DOOCS/MAIN:output',
           'SPB_XTD9_XGM/DOOCS/MAIN:output'})
```

Get data by train

[6]:
```
for tid, data in f.trains():
    print("Processing train", tid)
    print("beam iyPos:", data['SA1_XTD2_XGM/DOOCS/MAIN']['beamPosition.iyPos.value'])

    break
```

```
Processing train 10000
beam iyPos: 0.0
```

[7]:
```
tid, data = f.train_from_id(10005)
data['FXE_XAD_GEC/CAM/CAMERA:daqOutput']['data.image.dims']
```

[7]:
```
array([1024,  255], dtype=uint64)
```

These are just a few of the ways to access data. The attributes and methods described below for run directories also work with individual files. We expect that it will normally make sense to access a run directory as a single object, rather than working with the files separately.

3.9.2 Run directories

An experimental run is recorded as a collection of files in a directory.

Another dummy example:

[8]:
```
!ls fxe_example_run/
```

```
RAW-R0450-DA01-S00000.h5   RAW-R0450-LPD04-S00000.h5  RAW-R0450-LPD10-S00000.h5
RAW-R0450-DA01-S00001.h5   RAW-R0450-LPD05-S00000.h5  RAW-R0450-LPD11-S00000.h5
```

(continues on next page)

(continued from previous page)

```
RAW-R0450-LPD00-S00000.h5  RAW-R0450-LPD06-S00000.h5  RAW-R0450-LPD12-S00000.h5
RAW-R0450-LPD01-S00000.h5  RAW-R0450-LPD07-S00000.h5  RAW-R0450-LPD13-S00000.h5
RAW-R0450-LPD02-S00000.h5  RAW-R0450-LPD08-S00000.h5  RAW-R0450-LPD14-S00000.h5
RAW-R0450-LPD03-S00000.h5  RAW-R0450-LPD09-S00000.h5  RAW-R0450-LPD15-S00000.h5
```

[9]:
```
from karabo_data import RunDirectory
run = RunDirectory('fxe_example_run/')
```

[10]:
```
run.files[:3]   # The objects for the individual files (see above)
```

[10]:
```
[FileAccess(<HDF5 file "RAW-R0450-LPD04-S00000.h5" (mode r)>),
 FileAccess(<HDF5 file "RAW-R0450-LPD11-S00000.h5" (mode r)>),
 FileAccess(<HDF5 file "RAW-R0450-LPD15-S00000.h5" (mode r)>)]
```

What devices were recording in this run?

Control devices are slow data, recording once per train. *Instrument* devices includes detector data, but also some other data sources such as cameras. They can have more than one reading per train.

[11]:
```
run.control_sources
```

[11]:
```
frozenset({'FXE_XAD_GEC/CAM/CAMERA',
           'FXE_XAD_GEC/CAM/CAMERA_NODATA',
           'SA1_XTD2_XGM/DOOCS/MAIN',
           'SPB_XTD9_XGM/DOOCS/MAIN'})
```

[12]:
```
run.instrument_sources
```

[12]:
```
frozenset({'FXE_DET_LPD1M-1/DET/0CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/10CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/11CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/12CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/13CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/14CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/15CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/1CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/2CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/3CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/4CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/5CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/6CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/7CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/8CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/9CH0:xtdf',
           'FXE_XAD_GEC/CAM/CAMERA:daqOutput',
           'FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput',
           'SA1_XTD2_XGM/DOOCS/MAIN:output',
           'SPB_XTD9_XGM/DOOCS/MAIN:output'})
```

Which trains are in this run?

[13]:
```
print(run.train_ids[:10])
```

```
[10000, 10001, 10002, 10003, 10004, 10005, 10006, 10007, 10008, 10009]
```

See the available keys for a given source:

[14]:
```
run.keys_for_source('SPB_XTD9_XGM/DOOCS/MAIN:output')
```

[14]:
```
{'data.intensityAUXTD',
 'data.intensitySigma.x_data',
 'data.intensitySigma.y_data',
 'data.intensityTD',
 'data.trainId',
 'data.xTD',
 'data.yTD'}
```

This collects data from across files, including detector data:

[15]:
```
for tid, data in run.trains():
    print("Processing train", tid)
    print("Detctor data module 0 shape:", data['FXE_DET_LPD1M-1/DET/0CH0:xtdf'][
↪'image.data'].shape)

    break  # Stop after the first train to keep the demo short
```

```
Processing train 10000
Detctor data module 0 shape: (128, 1, 256, 256)
```

Train IDs are meant to be globally unique (although there were some glitches with this in the past). A train index is only within this run.

[16]:
```
tid, data = run.train_from_id(10005)
tid, data = run.train_from_index(5)
```

Series data to pandas

Data which holds a single number per train (or per pulse) can be extracted to as *series* (individual columns) and *dataframes* (tables) for pandas, a widely-used tool for data manipulation.

`karabo_data` chains sequence files, which contain successive data from the same source. In this example, trains 10000–10399 are in one sequence file (`...DA01-S00000.h5`), and 10400–10479 are in another (`...DA01-S00001.h5`). They are concatenated into one series:

[17]:
```
ixPos = run.get_series('SA1_XTD2_XGM/DOOCS/MAIN', 'beamPosition.ixPos.value')
ixPos.tail(10)
```

[17]:
```
trainId
10470    0.0
10471    0.0
10472    0.0
10473    0.0
10474    0.0
10475    0.0
10476    0.0
10477    0.0
10478    0.0
10479    0.0
Name: SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.ixPos, dtype: float32
```

To extract a dataframe, you can select interesting data fields with *glob* syntax, as often used for selecting files on Unix platforms.

- `[abc]`: one character, a/b/c
- `?`: any one character

- `*`: any sequence of characters

[18]:
```
run.get_dataframe(fields=[("*_XGM/*", "*.i[xy]Pos")])
```

[18]:

trainId	SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.ixPos
10000	0.0
10001	0.0
10002	0.0
10003	0.0
10004	0.0
10005	0.0
10006	0.0
10007	0.0
10008	0.0
10009	0.0
10010	0.0
10011	0.0
10012	0.0
10013	0.0
10014	0.0
10015	0.0
10016	0.0
10017	0.0
10018	0.0
10019	0.0
10020	0.0
10021	0.0
10022	0.0
10023	0.0
10024	0.0
10025	0.0
10026	0.0
10027	0.0
10028	0.0
10029	0.0
...	...
10450	0.0
10451	0.0
10452	0.0
10453	0.0
10454	0.0
10455	0.0
10456	0.0
10457	0.0
10458	0.0
10459	0.0
10460	0.0
10461	0.0
10462	0.0
10463	0.0
10464	0.0
10465	0.0
10466	0.0
10467	0.0
10468	0.0
10469	0.0
10470	0.0

(continues on next page)

(continued from previous page)

```
10471                                                 0.0
10472                                                 0.0
10473                                                 0.0
10474                                                 0.0
10475                                                 0.0
10476                                                 0.0
10477                                                 0.0
10478                                                 0.0
10479                                                 0.0

         SA1_XTD2_XGM/DOOCS/MAIN/beamPosition.iyPos  \
trainId
10000                                                 0.0
10001                                                 0.0
10002                                                 0.0
10003                                                 0.0
10004                                                 0.0
10005                                                 0.0
10006                                                 0.0
10007                                                 0.0
10008                                                 0.0
10009                                                 0.0
10010                                                 0.0
10011                                                 0.0
10012                                                 0.0
10013                                                 0.0
10014                                                 0.0
10015                                                 0.0
10016                                                 0.0
10017                                                 0.0
10018                                                 0.0
10019                                                 0.0
10020                                                 0.0
10021                                                 0.0
10022                                                 0.0
10023                                                 0.0
10024                                                 0.0
10025                                                 0.0
10026                                                 0.0
10027                                                 0.0
10028                                                 0.0
10029                                                 0.0
...                                                   ...
10450                                                 0.0
10451                                                 0.0
10452                                                 0.0
10453                                                 0.0
10454                                                 0.0
10455                                                 0.0
10456                                                 0.0
10457                                                 0.0
10458                                                 0.0
10459                                                 0.0
10460                                                 0.0
10461                                                 0.0
10462                                                 0.0
10463                                                 0.0
```

(continues on next page)

(continued from previous page)

```
10464                                                 0.0
10465                                                 0.0
10466                                                 0.0
10467                                                 0.0
10468                                                 0.0
10469                                                 0.0
10470                                                 0.0
10471                                                 0.0
10472                                                 0.0
10473                                                 0.0
10474                                                 0.0
10475                                                 0.0
10476                                                 0.0
10477                                                 0.0
10478                                                 0.0
10479                                                 0.0

         SPB_XTD9_XGM/DOOCS/MAIN/beamPosition.ixPos  \
trainId
10000                                                 0.0
10001                                                 0.0
10002                                                 0.0
10003                                                 0.0
10004                                                 0.0
10005                                                 0.0
10006                                                 0.0
10007                                                 0.0
10008                                                 0.0
10009                                                 0.0
10010                                                 0.0
10011                                                 0.0
10012                                                 0.0
10013                                                 0.0
10014                                                 0.0
10015                                                 0.0
10016                                                 0.0
10017                                                 0.0
10018                                                 0.0
10019                                                 0.0
10020                                                 0.0
10021                                                 0.0
10022                                                 0.0
10023                                                 0.0
10024                                                 0.0
10025                                                 0.0
10026                                                 0.0
10027                                                 0.0
10028                                                 0.0
10029                                                 0.0
...                                                   ...
10450                                                 0.0
10451                                                 0.0
10452                                                 0.0
10453                                                 0.0
10454                                                 0.0
10455                                                 0.0
10456                                                 0.0
```

(continues on next page)

(continued from previous page)

```
10457                                                 0.0
10458                                                 0.0
10459                                                 0.0
10460                                                 0.0
10461                                                 0.0
10462                                                 0.0
10463                                                 0.0
10464                                                 0.0
10465                                                 0.0
10466                                                 0.0
10467                                                 0.0
10468                                                 0.0
10469                                                 0.0
10470                                                 0.0
10471                                                 0.0
10472                                                 0.0
10473                                                 0.0
10474                                                 0.0
10475                                                 0.0
10476                                                 0.0
10477                                                 0.0
10478                                                 0.0
10479                                                 0.0

         SPB_XTD9_XGM/DOOCS/MAIN/beamPosition.iyPos
trainId
10000                                                 0.0
10001                                                 0.0
10002                                                 0.0
10003                                                 0.0
10004                                                 0.0
10005                                                 0.0
10006                                                 0.0
10007                                                 0.0
10008                                                 0.0
10009                                                 0.0
10010                                                 0.0
10011                                                 0.0
10012                                                 0.0
10013                                                 0.0
10014                                                 0.0
10015                                                 0.0
10016                                                 0.0
10017                                                 0.0
10018                                                 0.0
10019                                                 0.0
10020                                                 0.0
10021                                                 0.0
10022                                                 0.0
10023                                                 0.0
10024                                                 0.0
10025                                                 0.0
10026                                                 0.0
10027                                                 0.0
10028                                                 0.0
10029                                                 0.0
...                                                   ...
```

(continues on next page)

(continued from previous page)

```
10450                                                0.0
10451                                                0.0
10452                                                0.0
10453                                                0.0
10454                                                0.0
10455                                                0.0
10456                                                0.0
10457                                                0.0
10458                                                0.0
10459                                                0.0
10460                                                0.0
10461                                                0.0
10462                                                0.0
10463                                                0.0
10464                                                0.0
10465                                                0.0
10466                                                0.0
10467                                                0.0
10468                                                0.0
10469                                                0.0
10470                                                0.0
10471                                                0.0
10472                                                0.0
10473                                                0.0
10474                                                0.0
10475                                                0.0
10476                                                0.0
10477                                                0.0
10478                                                0.0
10479                                                0.0

[480 rows x 4 columns]
```

Labelled arrays

Data with extra dimensions can be handled as xarray labelled arrays. These are a wrapper around Numpy arrays with indexes which can be used to align them and select data.

[19]:
```
xtd2_intensity = run.get_array('SA1_XTD2_XGM/DOOCS/MAIN:output', 'data.intensityTD',␣
↪extra_dims=['pulseID'])
xtd2_intensity
```

[19]:
```
<xarray.DataArray (trainId: 480, pulseID: 1000)>
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 10000 10001 10002 10003 ... 10477 10478 10479
Dimensions without coordinates: pulseID
```

Here's a brief example of using xarray to align the data and select by train ID. See the examples in the xarray docs for more on what it can do.

In this example data, all the data sources have the same range of train IDs, so aligning them doesn't change anything. In real data, devices may miss some trains that other devices did record.

[20]:
```
import xarray as xr
xtd9_intensity = run.get_array('SPB_XTD9_XGM/DOOCS/MAIN:output', 'data.intensityTD',␣
↪extra_dims=['pulseID'])

# Align two arrays, keep only trains which they both have data for:
xtd2_intensity, xtd9_intensity = xr.align(xtd2_intensity, xtd9_intensity, join='inner
↪')

# Select data for a single train by train ID:
xtd2_intensity.sel(trainId=10004)

# Select data from a range of train IDs.
# This includes the end value, unlike normal Python indexing
xtd2_intensity.loc[10004:10006]
```

[20]:
```
<xarray.DataArray (trainId: 3, pulseID: 1000)>
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 10004 10005 10006
Dimensions without coordinates: pulseID
```

You can also specify a region of interest from an array to load only part of the data:

[21]:
```
from karabo_data import by_index

# Select the first 5 trains in this run:
sel = run.select_trains(by_index[:5])

# Get the whole of this array:
arr = sel.get_array('FXE_XAD_GEC/CAM/CAMERA:daqOutput', 'data.image.pixels')
print("Whole array shape:", arr.shape)

# Get a region of interest
arr2 = sel.get_array('FXE_XAD_GEC/CAM/CAMERA:daqOutput', 'data.image.pixels', roi=by_
↪index[100:200, :512])
print("ROI array shape:", arr2.shape)
```

```
Whole array shape: (5, 255, 1024)
ROI array shape: (5, 100, 512)
```

3.9.3 General information

`karabo_data` provides a few ways to get general information about what's in data files. First, from Python code:

[22]:
```
run.info()
```

```
# of trains:    480
Duration:       0:00:47.900000
First train ID: 10000
Last train ID:  10479

16 detector modules (FXE_DET_LPD1M-1)
  e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
```

(continues on next page)

(continued from previous page)

```
  128 frames per train, 61440 total frames

4 instrument sources (excluding detectors):
  - FXE_XAD_GEC/CAM/CAMERA:daqOutput
  - FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput
  - SA1_XTD2_XGM/DOOCS/MAIN:output
  - SPB_XTD9_XGM/DOOCS/MAIN:output

4 control sources:
  - FXE_XAD_GEC/CAM/CAMERA
  - FXE_XAD_GEC/CAM/CAMERA_NODATA
  - SA1_XTD2_XGM/DOOCS/MAIN
  - SPB_XTD9_XGM/DOOCS/MAIN
```

[23]:
```
run.detector_info('FXE_DET_LPD1M-1/DET/0CH0:xtdf')
```

[23]:
```
{'dims': (256, 256), 'frames_per_train': 128, 'total_frames': 61440}
```

The `lsxfel` command provides similar information at the command line:

[24]:
```
!lsxfel fxe_example_run/RAW-R0450-LPD00-S00000.h5
```

```
RAW-R0450-LPD00-S00000.h5 : Raw detector data from LPD module 00
480 trains

256 × 256 pixels
128 frames per train, 61440 total
```

[25]:
```
!lsxfel fxe_example_run/RAW-R0450-DA01-S00000.h5
```

```
RAW-R0450-DA01-S00000.h5 : Aggregated data
400 trains

4 instrument sources
  -  FXE_XAD_GEC/CAM/CAMERA:daqOutput
  -  FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput
  -  SA1_XTD2_XGM/DOOCS/MAIN:output
  -  SPB_XTD9_XGM/DOOCS/MAIN:output

4 control sources
  -  FXE_XAD_GEC/CAM/CAMERA
  -  FXE_XAD_GEC/CAM/CAMERA_NODATA
  -  SA1_XTD2_XGM/DOOCS/MAIN
  -  SPB_XTD9_XGM/DOOCS/MAIN
```

[26]:
```
!lsxfel fxe_example_run
```

```
fxe_example_run : Run directory

# of trains:    480
Duration:       0:00:47.900000
First train ID: 10000
Last train ID:  10479

16 detector modules (FXE_DET_LPD1M-1)
```

(continues on next page)

(continued from previous page)

```
  e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
  128 frames per train, 61440 total frames

4 instrument sources (excluding detectors):
  - FXE_XAD_GEC/CAM/CAMERA:daqOutput
  - FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput
  - SA1_XTD2_XGM/DOOCS/MAIN:output
  - SPB_XTD9_XGM/DOOCS/MAIN:output

4 control sources:
  - FXE_XAD_GEC/CAM/CAMERA
  - FXE_XAD_GEC/CAM/CAMERA_NODATA
  - SA1_XTD2_XGM/DOOCS/MAIN
  - SPB_XTD9_XGM/DOOCS/MAIN
```

3.10 Accessing LPD data

The Large Pixel Detector (LPD) is made of 16 modules which record data separately. `karabo_data` includes convenient interfaces to access this data together.

This example stands by itself, but if you need more generic access to the data, please see *Reading data with karabo_data*.

First, let's load a run containing LPD data:

[1]:
```
from karabo_data import RunDirectory, by_index

run = RunDirectory('fxe_example_run/')
# Using only the first three trains to keep this example light:
run = run.select_trains(by_index[:3])

run.instrument_sources
```

[1]:
```
frozenset({'FXE_DET_LPD1M-1/DET/0CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/10CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/11CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/12CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/13CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/14CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/15CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/1CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/2CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/3CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/4CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/5CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/6CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/7CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/8CH0:xtdf',
           'FXE_DET_LPD1M-1/DET/9CH0:xtdf',
           'FXE_XAD_GEC/CAM/CAMERA:daqOutput',
           'FXE_XAD_GEC/CAM/CAMERA_NODATA:daqOutput',
           'SA1_XTD2_XGM/DOOCS/MAIN:output',
           'SPB_XTD9_XGM/DOOCS/MAIN:output'})
```

Normal access methods give us each module separately:

[2]:
```
data_module0 = run.get_array('FXE_DET_LPD1M-1/DET/0CH0:xtdf', 'image.data')
data_module0.shape
```

[2]:
```
(384, 1, 256, 256)
```

The class `karabo_data.components.LPD1M` can piece these together:

[3]:
```
from karabo_data.components import LPD1M
lpd = LPD1M(run)
lpd
```

[3]:
```
<LPD1M: Data interface for detector 'FXE_DET_LPD1M-1' with 16 modules>
```

[4]:
```
image_data = lpd.get_array('image.data')
print("Data shape:", image_data.shape)
print("Dimensions:", image_data.dims)
```

```
Data shape: (16, 3, 128, 256, 256)
Dimensions: ('module', 'train', 'pulse', 'slow_scan', 'fast_scan')
```

Note: This class pulls the data together, but it doesn't know how the modules are physically arranged, so it can't produce a detector image. Other examples show how to use detector geometry to produce images.

You can also select only certain modules of the detector. For example, modules 2 (Q1M3), 7 (Q2M4), 8 (Q3M1) and 13 (Q4M2) are the four modules around the center of the detector:

[5]:
```
lpd = LPD1M(run, modules=[2, 7, 8, 13])
image_data = lpd.get_array('image.data')
print("Data shape:", image_data.shape)
print("Dimensions:", image_data.dims)

print()
print("Data for one pulse:")
print(image_data.sel(train=10000, pulse=0))
```

```
Data shape: (4, 3, 128, 256, 256)
Dimensions: ('module', 'train', 'pulse', 'slow_scan', 'fast_scan')

Data for one pulse:
<xarray.DataArray (module: 4, slow_scan: 256, fast_scan: 256)>
array([[[0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0],
        ...,
        [0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0]],

       [[0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0],
        ...,
        [0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0]],

       [[0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0],
        ...,
        [0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0]],

       [[0, 0, ..., 0, 0],
```

(continues on next page)

(continued from previous page)

```
        [0, 0, ..., 0, 0],
        ...,
        [0, 0, ..., 0, 0],
        [0, 0, ..., 0, 0]]], dtype=uint16)
Coordinates:
    pulse    uint64 0
    train    uint64 10000
  * module   (module) int64 2 7 8 13
Dimensions without coordinates: slow_scan, fast_scan
```

The returned array is an *xarray* object with labelled axes. See Indexing and selecting data in the xarray docs for more on what you can do with it.

This interface also supports iterating train-by-train through detector data, giving labelled arrays again:

[6]:
```
for tid, train_data in lpd.trains(pulses=by_index[:16]):
    print("Train", tid)
    print("Keys in data:", sorted(train_data.keys()))
    print("Image data shape:", train_data['image.data'].shape)
    print()
```

```
Train 10000
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
 ↪'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
 ↪minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
 ↪'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
 ↪'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
 ↪'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)

Train 10001
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
 ↪'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
 ↪minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
 ↪'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
 ↪'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
 ↪'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)

Train 10002
Keys in data: ['detector.data', 'detector.trainId', 'header.dataId', 'header.linkId',
 ↪'header.magicNumberBegin', 'header.majorTrainFormatVersion', 'header.
 ↪minorTrainFormatVersion', 'header.pulseCount', 'header.reserved', 'header.trainId',
 ↪'image.cellId', 'image.data', 'image.length', 'image.pulseId', 'image.status',
 ↪'image.trainId', 'trailer.checksum', 'trailer.magicNumberEnd', 'trailer.status',
 ↪'trailer.trainId']
Image data shape: (4, 1, 16, 256, 256)
```

3.11 Assembling detector data into images

The X-ray detectors at XFEL are made up of a number of small pieces. To get an image from the data, or analyse it spatially, we need to know where each piece is located.

This example reassembles some commissioning data from LPD, a detector which has 4 quadrants, 16 modules, and 256 tiles. Elements (especially the quadrants) can be repositioned; talk to the detector group to ensure that you have the right geometry information for your data.

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import h5py

from karabo_data import RunDirectory, stack_detector_data
from karabo_data.geometry2 import LPD_1MGeometry
```

[2]:
```
run = RunDirectory('/gpfs/exfel/exp/FXE/201830/p900020/proc/r0221/')
run.info()
```

```
# of trains:    513
Duration:       0:00:51.200000
First train ID: 54861753
Last train ID:  54862265

14 detector modules (FXE_DET_LPD1M-1)
  e.g. module FXE_DET_LPD1M-1 0 : 256 x 256 pixels
  128 frames per train, 39040 total frames

0 instrument sources (excluding detectors):

0 control sources:
```

[3]:
```
# Find a train with some data in
empty = np.asarray([])
for tid, train_data in run.trains():
    module_imgs = sum(d.get('image.data', empty).shape[0] for d in train_data.
↪values())
    if module_imgs:
        print(tid, module_imgs)
        break
```

```
54861797 1792
```

[4]:
```
tid, train_data = run.train_from_id(54861797)
print(tid)
for dev in sorted(train_data.keys()):
    print(dev, end='\t')
    try:
        print(train_data[dev]['image.data'].shape)
    except KeyError:
        print("No image.data")
```

```
54861797
FXE_DET_LPD1M-1/DET/0CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/10CH0:xtdf        (128, 256, 256)
```

(continues on next page)

(continued from previous page)

```
FXE_DET_LPD1M-1/DET/11CH0:xtdf        (128, 256, 256)
FXE_DET_LPD1M-1/DET/12CH0:xtdf        (128, 256, 256)
FXE_DET_LPD1M-1/DET/13CH0:xtdf        (128, 256, 256)
FXE_DET_LPD1M-1/DET/14CH0:xtdf        (128, 256, 256)
FXE_DET_LPD1M-1/DET/15CH0:xtdf        (128, 256, 256)
FXE_DET_LPD1M-1/DET/1CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/2CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/3CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/4CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/6CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/8CH0:xtdf (128, 256, 256)
FXE_DET_LPD1M-1/DET/9CH0:xtdf (128, 256, 256)
```

Extract the detector images into a single Numpy array:

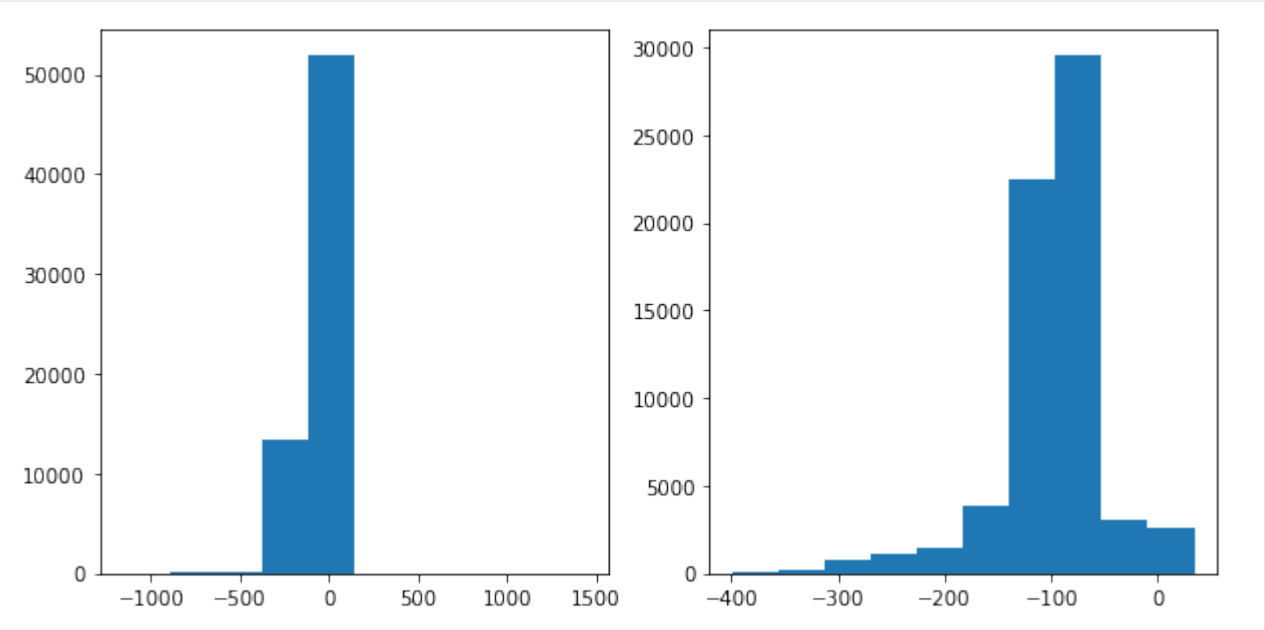
[5]:
```
modules_data = stack_detector_data(train_data, 'image.data')
modules_data.shape
```

[5]:
```
(128, 16, 256, 256)
```

To show the images, we sometimes need to 'clip' extreme high and low values, otherwise the colour map makes everything else the same colour.

[6]:
```
def clip(array, min=-10000, max=10000):
    x = array.copy()
    finite = np.isfinite(x)
    # Suppress warnings comparing numbers to nan
    with np.errstate(invalid='ignore'):
        x[finite & (x < min)] = np.nan
        x[finite & (x > max)] = np.nan
    return x
```

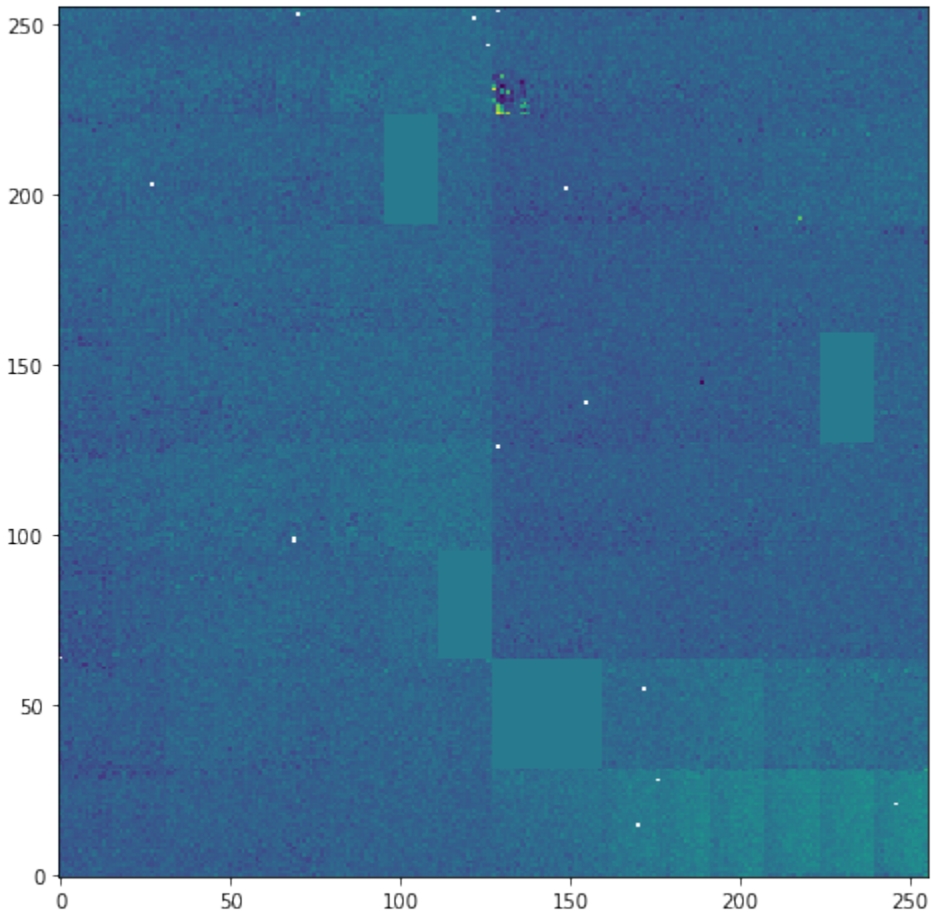
[7]:
```
plt.figure(figsize=(10, 5))

a = modules_data[5][2]
plt.subplot(1, 2, 1).hist(a[np.isfinite(a)])

a = clip(a, min=-400, max=400)
plt.subplot(1, 2, 2).hist(a[np.isfinite(a)]);
```



Let's look at the iamge from a single module. You can see where it's divided up into tiles:

[8]:
```
plt.figure(figsize=(8, 8))
clipped_mod = clip(modules_data[10][2], -400, 500)
plt.imshow(clipped_mod, origin='lower')
```

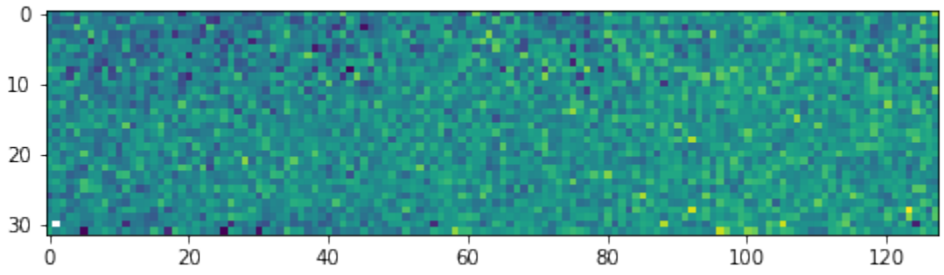
[8]:
```
<matplotlib.image.AxesImage at 0x2b611fe49390>
```



Here's a single tile:

[9]:
```
splitted = LPD_1MGeometry.split_tiles(clipped_mod)
plt.figure(figsize=(8, 8))
plt.imshow(splitted[11])
```

[9]:
```
<matplotlib.image.AxesImage at 0x2b611feb5080>
```



Load the geometry from a file, along with the quadrant positions used here.

In the future, geometry information will be stored in the calibration catalogue.

[10]:
```
# From March 18; converted to XFEL standard coordinate directions
quadpos = [(11.4, 299), (-11.5, 8), (254.5, -16), (278.5, 275)]  # mm
```

(continues on next page)

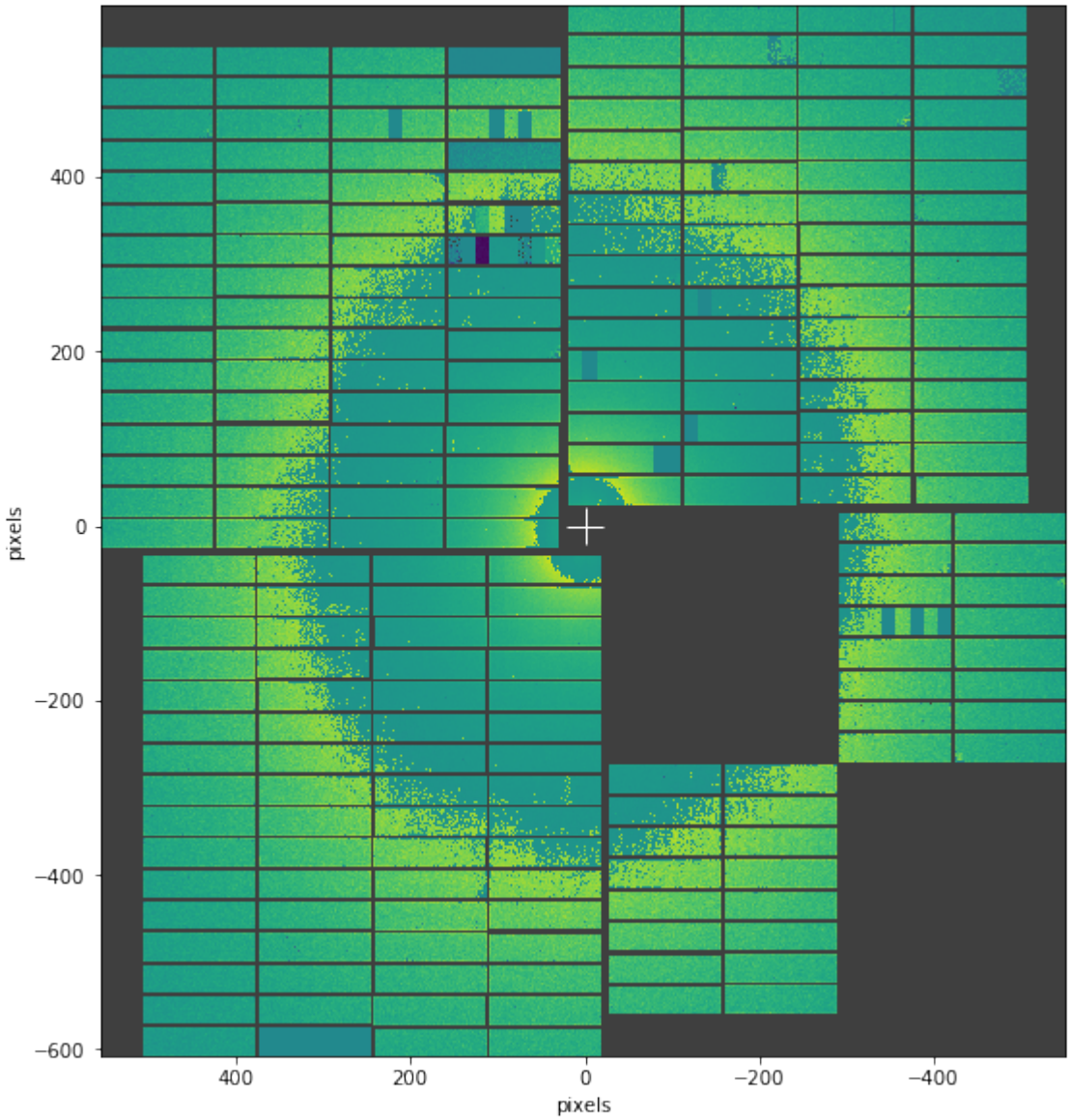
(continued from previous page)

```
geom = LPD_1MGeometry.from_h5_file_and_quad_positions('lpd_mar_18_axesfixed.h5',␣
↪quadpos)
```

Reassemble and show a detector image using the geometry:

[11]:
```
geom.plot_data_fast(clip(modules_data[12], max=5000))
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b611ff3de48>
```



Reassemble detector data into a numpy array for further analysis. The areas without data have the special value ``*nan*`` to mark them as missing.

[12]:
```
res, centre = geom.position_modules_fast(modules_data)
print(res.shape)
plt.figure(figsize=(8, 8))
```

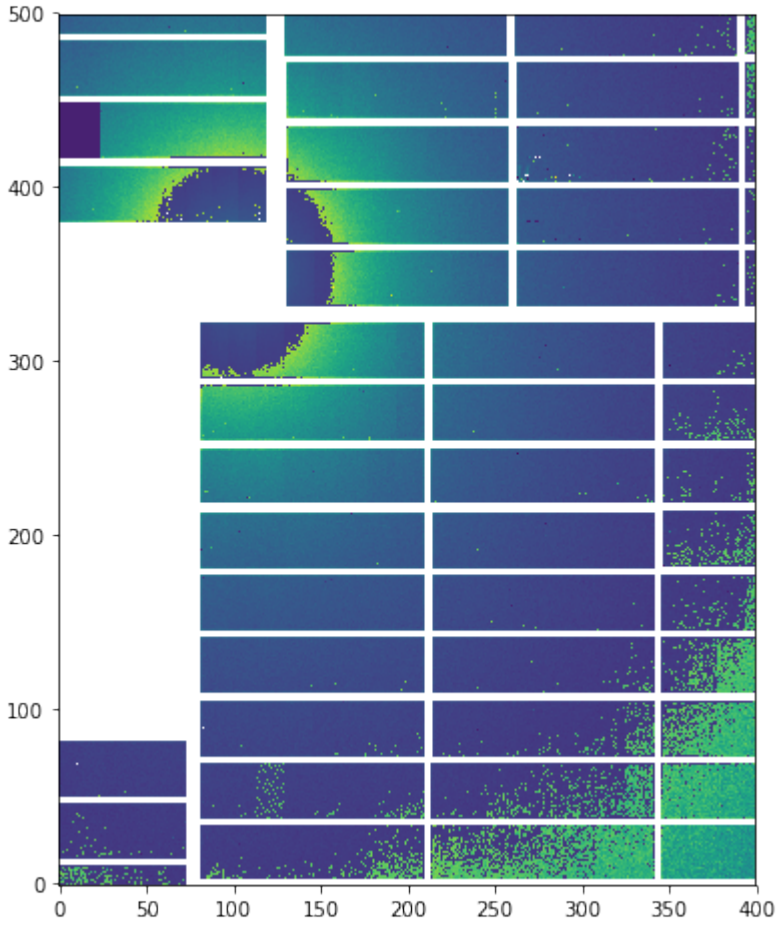
(continues on next page)

(continued from previous page)

```
plt.imshow(clip(res[12, 250:750, 450:850], min=-400, max=5000), origin='lower')
```

```
(128, 1203, 1105)
```

[12]:
```
<matplotlib.image.AxesImage at 0x2b60ec9f4160>
```



3.12 Examining detector geometry

The *Applying geometry notebook* shows how to use detector geometry to assemble data into an image. We can also examine geometry information without any data, to check for problems.

[1]:
```
%matplotlib inline
from itertools import product
import numpy as np
import matplotlib.pyplot as plt
import h5py

from karabo_data import RunDirectory
from karabo_data.geometry2 import LPD_1MGeometry
```

This is some geometry for LPD. You can see that Q2M2 is 'missing' - in fact all its tiles are showing up in Q2M4.

Each module has tiles 1-16 running anticlockwise from the top left (looking into the beam). To make it visually clearer,

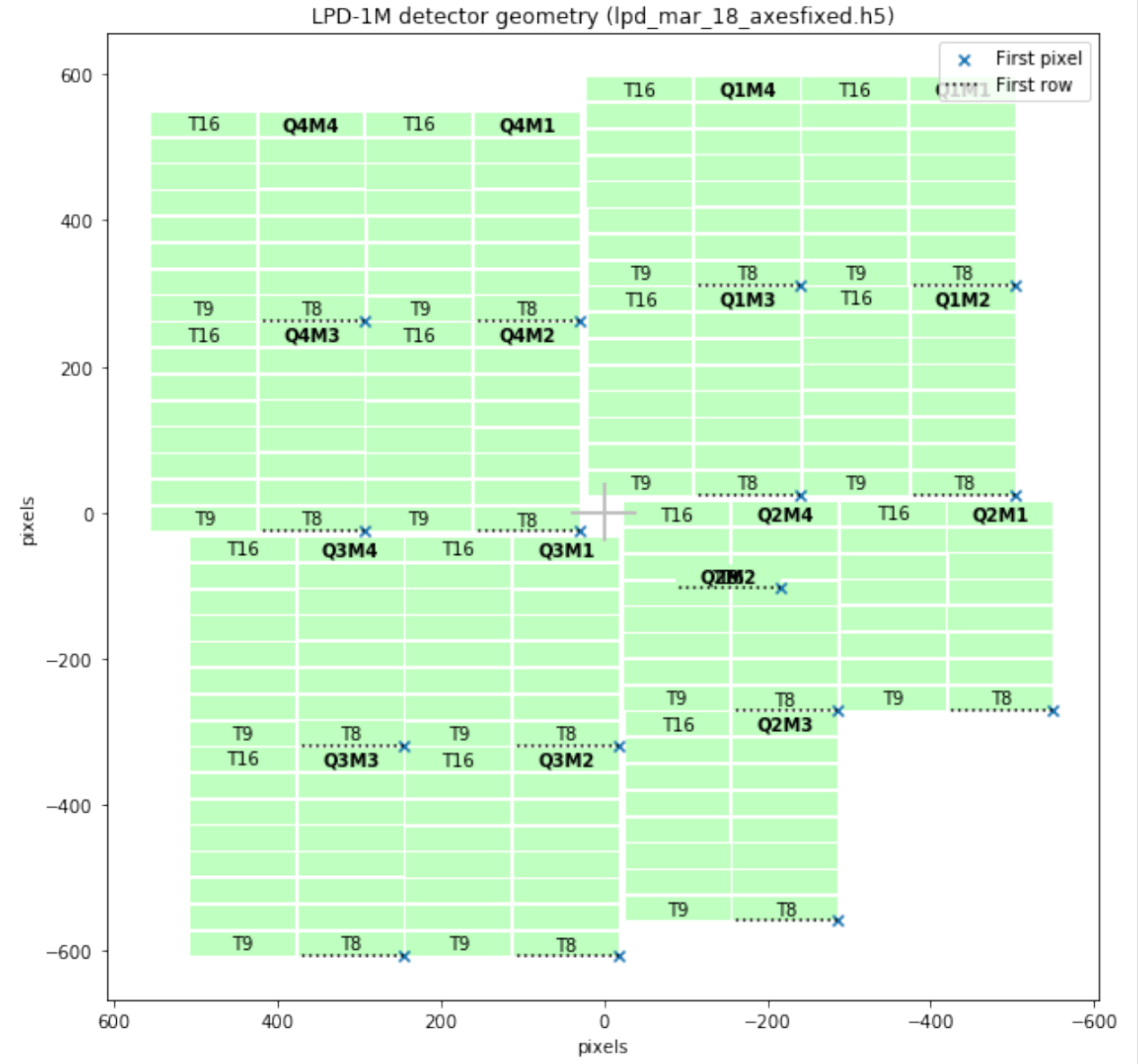
only three corner tiles of each module are numbered.

Here we are loading the geometry from a file, but in the future it will be possible to get the information from the calibration database directly.

[2]:
```
# From March 18; converted to XFEL standard coordinate directions
quadpos = [(11.4, 299), (-11.5, 8), (254.5, -16), (278.5, 275)]  # mm
geom = LPD_1MGeometry.from_h5_file_and_quad_positions('lpd_mar_18_axesfixed.h5',␣
↪quadpos)
```

[3]:
```
geom.inspect()
```

[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2ba73fa92a58>
```



3.13 Detector geometry for AGIPD

The AGIPD detector, which is already in use at the SPB experiment, consists of 16 modules of 512×128 pixels each. Each module is further divided into 8 ASICs.

To view or analyse detector data, we need to apply geometry to find the positions of pixels.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

from karabo_data import RunDirectory, stack_detector_data
from karabo_data.geometry2 import AGIPD_1MGeometry
```

Fetch AGIPD detector data for one pulse to test with:

[2]:
```
run = RunDirectory('/gpfs/exfel/exp/SPB/201831/p900039/proc/r0273/')
```

[3]:
```
tid, train_data = run.select('*/DET/*', 'image.data').train_from_index(60)
```

[4]:
```
stacked = stack_detector_data(train_data, 'image.data')
stacked_pulse = stacked[10]
stacked_pulse.shape
```

[4]:
```
(16, 512, 128)
```

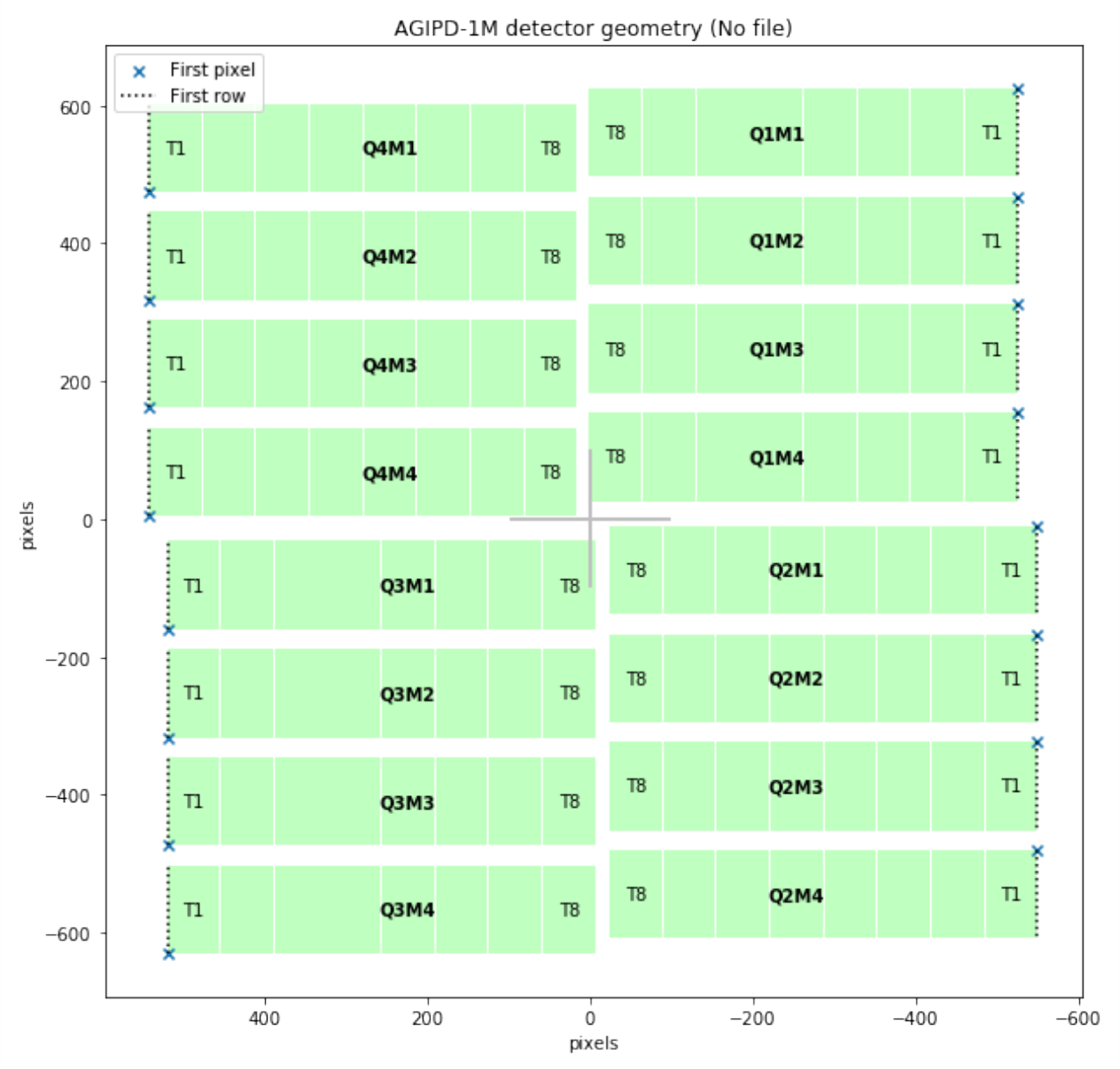
Generate a simple geometry given the (x, y) coordinates of the first pixel in the first module of each quadrant, in pixel units relative to the centre, where the beam passes through the detector.

There are also methods to load and save CrystFEL format geometry files.

[5]:
```
geom = AGIPD_1MGeometry.from_quad_positions(quad_pos=[
        (-525, 625),
        (-550, -10),
        (520, -160),
        (542.5, 475),
    ])
```

[6]:
```
geom.inspect()
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b1dae4100f0>
```



The pixels are not necessarily all aligned, so precisely assembling data in a 2D array requires interpolation, which is slow:

[7]:
```
%%time
data, centre_yx = geom.position_modules_interpolate(stacked_pulse)
print(data.shape)
```

```
(1258, 1094)
CPU times: user 10.9 s, sys: 1.08 s, total: 11.9 s
Wall time: 6 s
```

But we know that the modules are closely aligned with the axes, so we can 'snap' the geometry to the grid and copy data more efficiently:

[8]:
```
%%time
data, centre_yx = geom.position_modules_fast(stacked_pulse)
```

(continues on next page)

(continued from previous page)

```
print(data.shape)
```

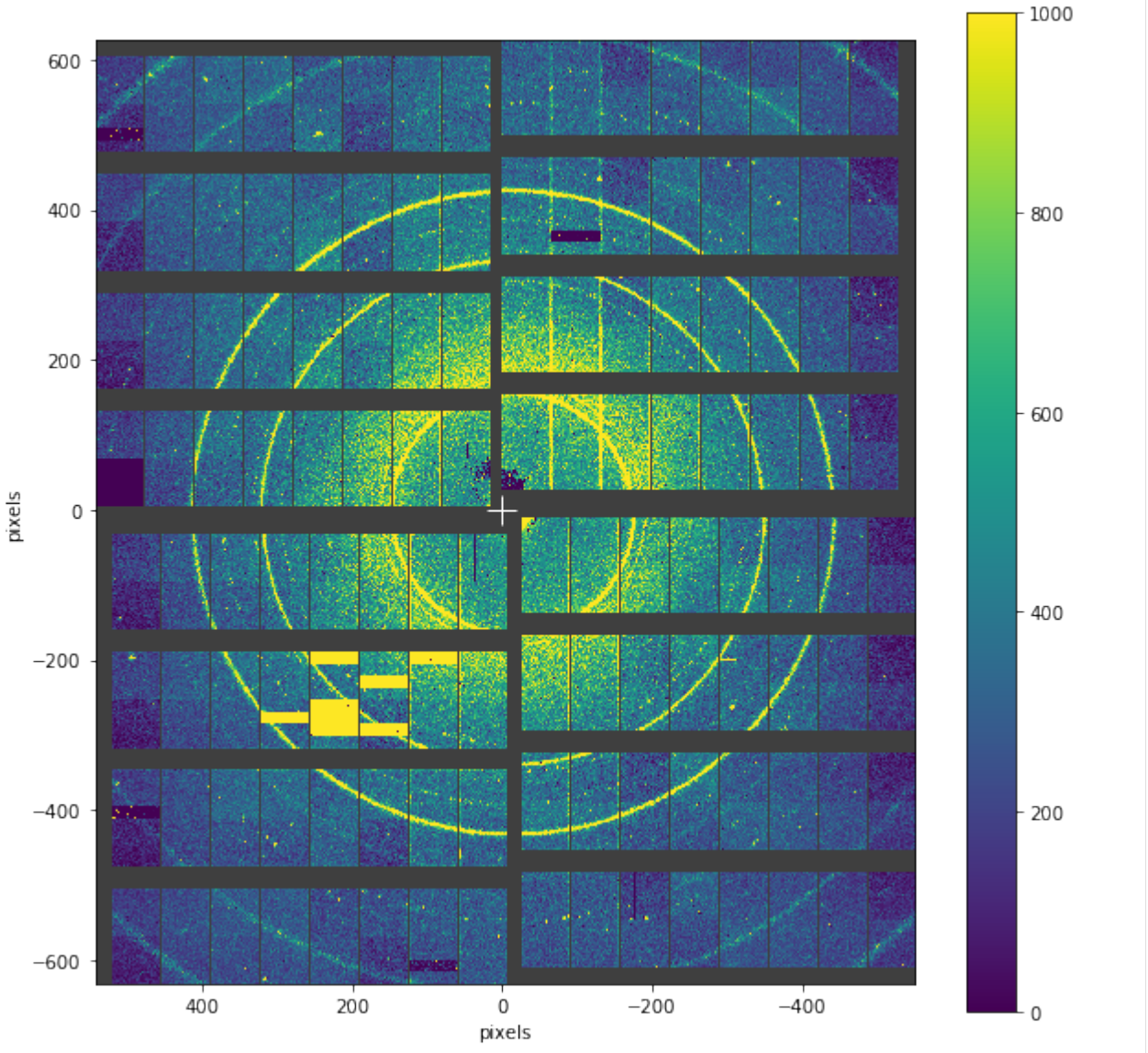
```
(1256, 1092)
CPU times: user 23.9 ms, sys: 9.73 ms, total: 33.6 ms
Wall time: 29.6 ms
```

3.13.1 Plot the detector image

Data can be directly plotted using the `plot_data_fast` method.

[9]:
```
geom.plot_data_fast(stacked_pulse, vmin=0, vmax=1000)
```

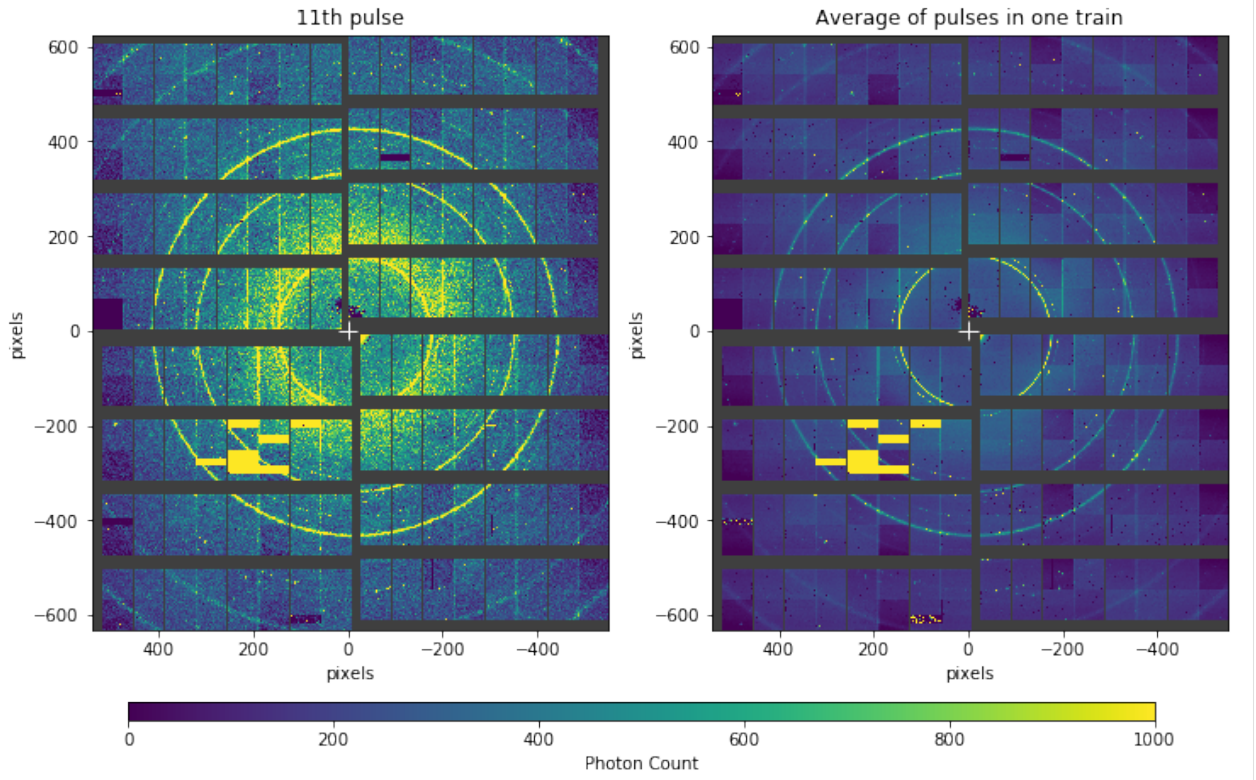
[9]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b1e67f828d0>
```



You can control the plot using keyword arguments for axis and colorbar. For example, to plot two images in the same figure:

[10]:
```
fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(12, 7.5))
ax_cbar = fig.add_axes([0.15, 0.08, 0.7, 0.02])  # Create extra axes for the colorbar

# Plot a single pulse in the left axes
geom.plot_data_fast(stacked_pulse, vmin=0, vmax=1000, ax=ax0, colorbar={
                            'cax': ax_cbar,
                            'shrink': 0.6,
                            'pad': 0.1,
                            'orientation': 'horizontal'
                        })
ax0.set_title('11th pulse')

# Label the colorbar associated with the first image
colorbar = ax0.images[0].colorbar
colorbar.set_label('Photon Count')

# Plot the average over all pulses on the right.
# Disable the colorbar because it's the same scale as the left image.
geom.plot_data_fast(stacked.mean(axis=0), vmin=0, vmax=1000, ax=ax1, colorbar=False)
ax1.set_title('Average of pulses in one train')
```

[10]:
```
Text(0.5, 1.0, 'Average of pulses in one train')
```



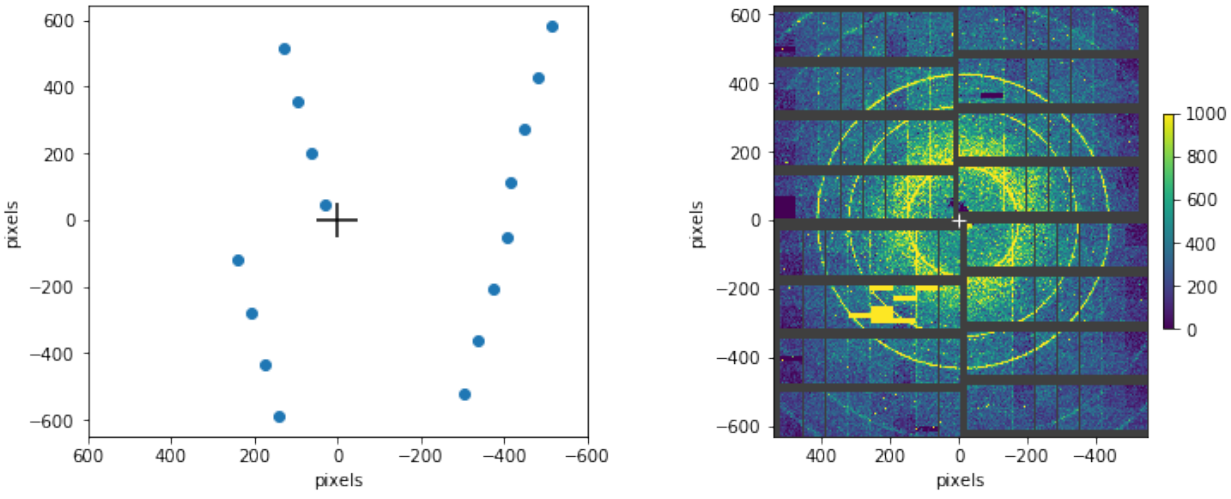
3.13.2 Converting array positions to physical positions

We can also convert array coordinates within the detector data into real (x, y, z) positions in metres.

[11]:
```
# Generate some array coordinates, one in each module
module_no = np.arange(0, 16)
# For AGIPD, slow-scan is the x dimension, increasing from the edges towards the␣
↪centre
slow_scan = np.linspace(10, 500, num=16)
fast_scan = np.full(fill_value=40.1, shape=16)  # Fixed y position in each module
```

[12]:
```
positions = geom.data_coords_to_positions(module_no, slow_scan, fast_scan)
print("positions.shape =", positions.shape)  # (point, x/y/z)

# Convert metres to pixel units to compare with plots above
px = positions[:, 0] / geom.pixel_size
py = positions[:, 1] / geom.pixel_size

fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(12, 6))

ax0.scatter(px, py)
ax0.set_xlabel('pixels')
ax0.set_ylabel('pixels')
ax0.hlines(0, -50, 50)  # Draw a cross at the origin
ax0.vlines(0, -50, 50)  #
ax0.set_xlim(600, -600)  # Invert x-axis to match plots above

# Display the image alongside it for comparison
geom.plot_data_fast(stacked_pulse, vmin=0, vmax=1000, ax=ax1,
                    colorbar={'shrink': 0.5, 'pad': 0.03})
fig.subplots_adjust(bottom=0.3, wspace=0.3)
```

```
positions.shape = (16, 3)
```



3.14 DSSC detector geometry

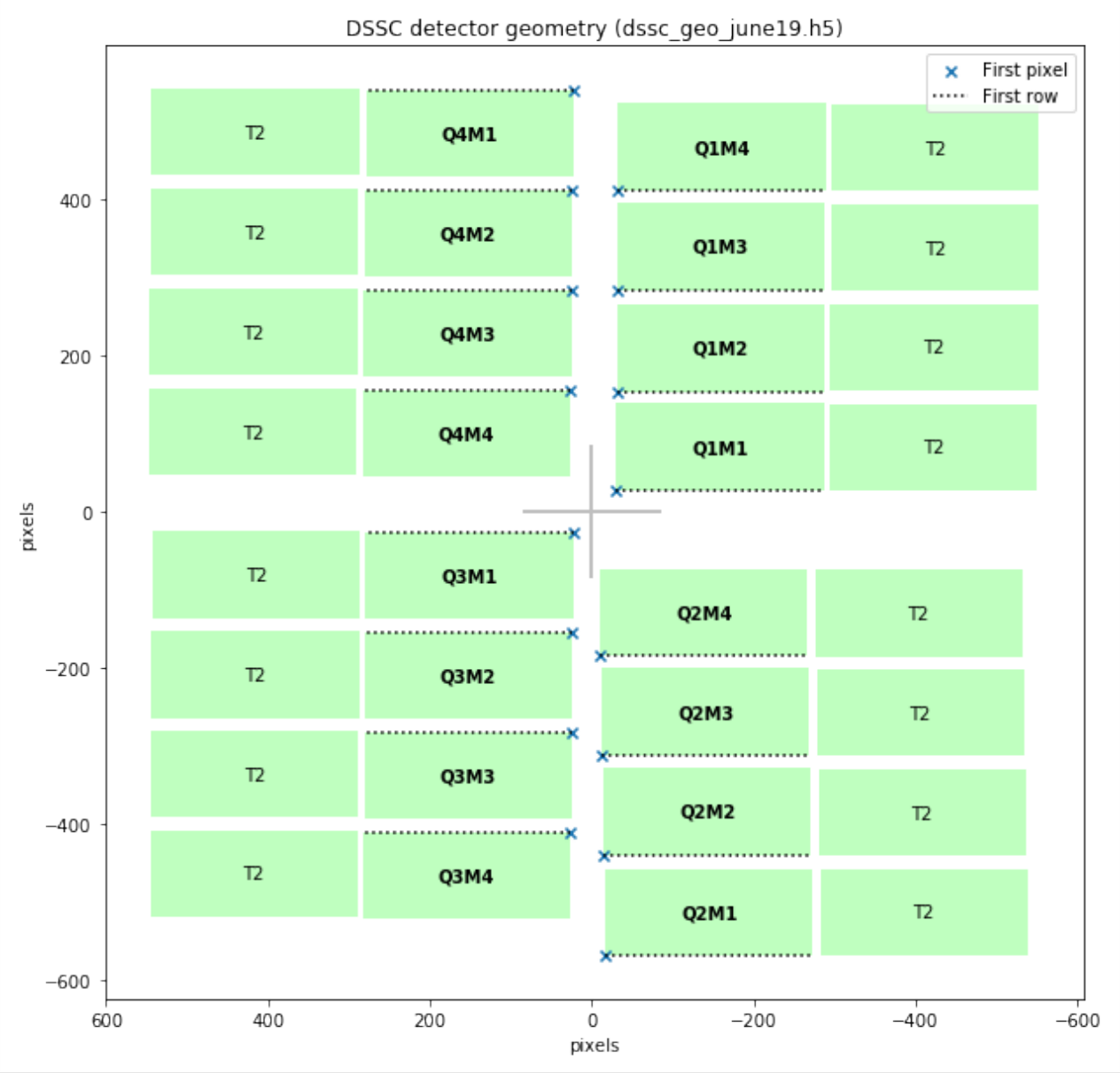
As of version 0.5, `karabo_data` has geometry code for the DSSC detector. This doesn't currently account for the hexagonal pixels of DSSC, but it's good enough for a preview of detector images.

[1]:
```
%matplotlib inline
from karabo_data.geometry2 import DSSC_1MGeometry
```

[2]:
```
# Made up numbers!
quad_pos = [
    (-130, 5),
    (-130, -125),
    (5, -125),
    (5, 5),
]
path = 'dssc_geo_june19.h5'

g = DSSC_1MGeometry.from_h5_file_and_quad_positions(path, quad_pos)
```

[3]:
```
g.inspect()
```

[3]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2ac10f8709b0>
```



[4]:
```
import numpy as np
import matplotlib.pyplot as plt
```

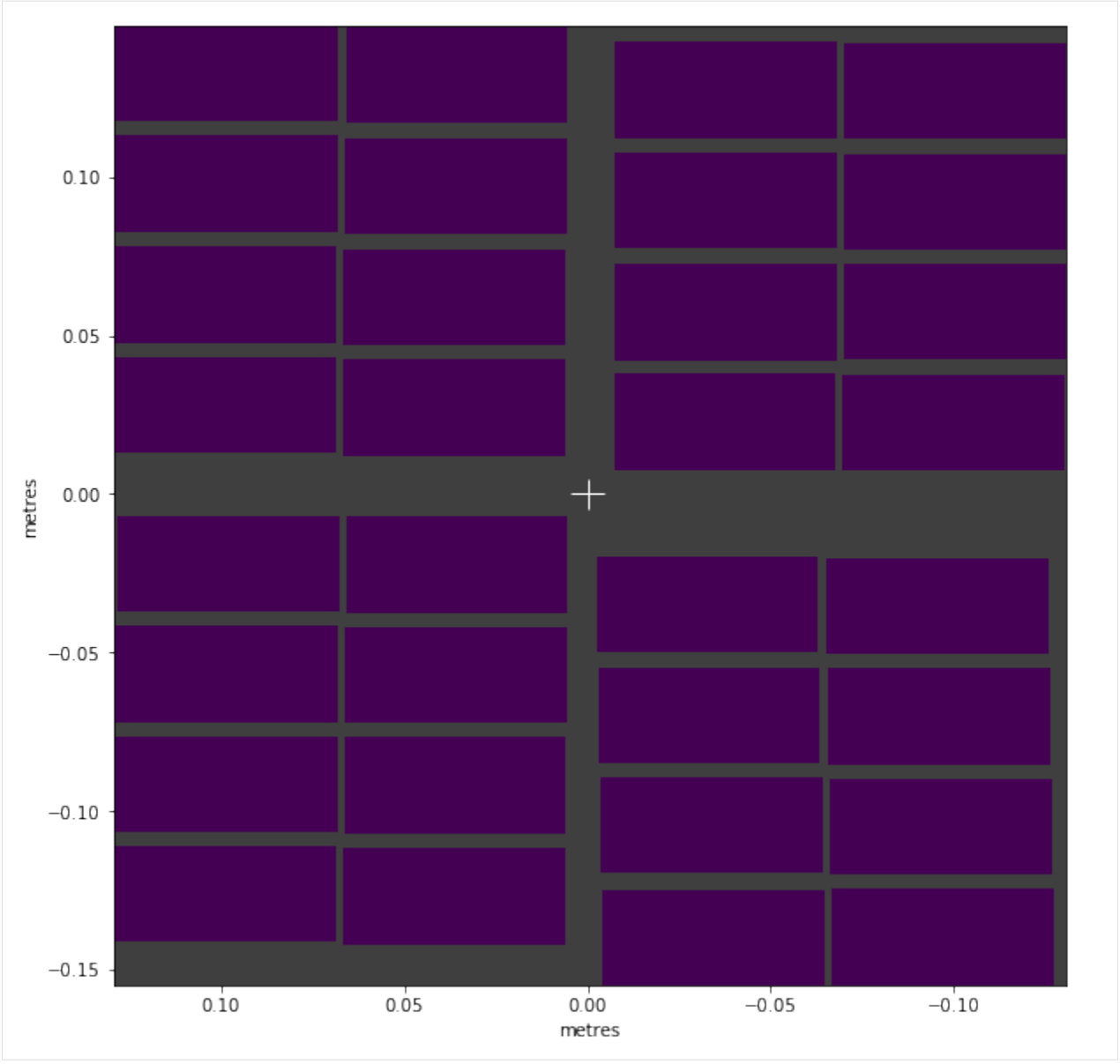
[5]:
```
g.expected_data_shape
```

[5]:
```
(16, 128, 512)
```

We'll use some empty data to demonstate assembling an image.

[6]:
```
a = np.zeros(g.expected_data_shape)
```

[7]:
```
g.plot_data_fast(a, axis_units='m');
```



Let's have a close up look at some pixels in Q1M1. `get_pixel_positions()` gives us pixel centres. `to_distortion_array()` gives pixel corners in a slightly different format, suitable for PyFAI.

PyFAI requires non-negative x and y coordinates. But we want to plot them along with the centre positions, so we pass `allow_negative_xy=True` to get comparable coordinates.

[8]:
```
pixel_pos = g.get_pixel_positions()
print("Pixel positions array shape:", pixel_pos.shape,
      "= (modules, slow_scan, fast_scan, x/y/z)")
q1m1_centres = pixel_pos[0]
cx = q1m1_centres[..., 0]
cy = q1m1_centres[..., 1]

distortn = g.to_distortion_array(allow_negative_xy=True)
print("Distortion array shape:", distortn.shape,
      "= (modules * slow_scan, fast_scan, corners, z/y/x)")
```

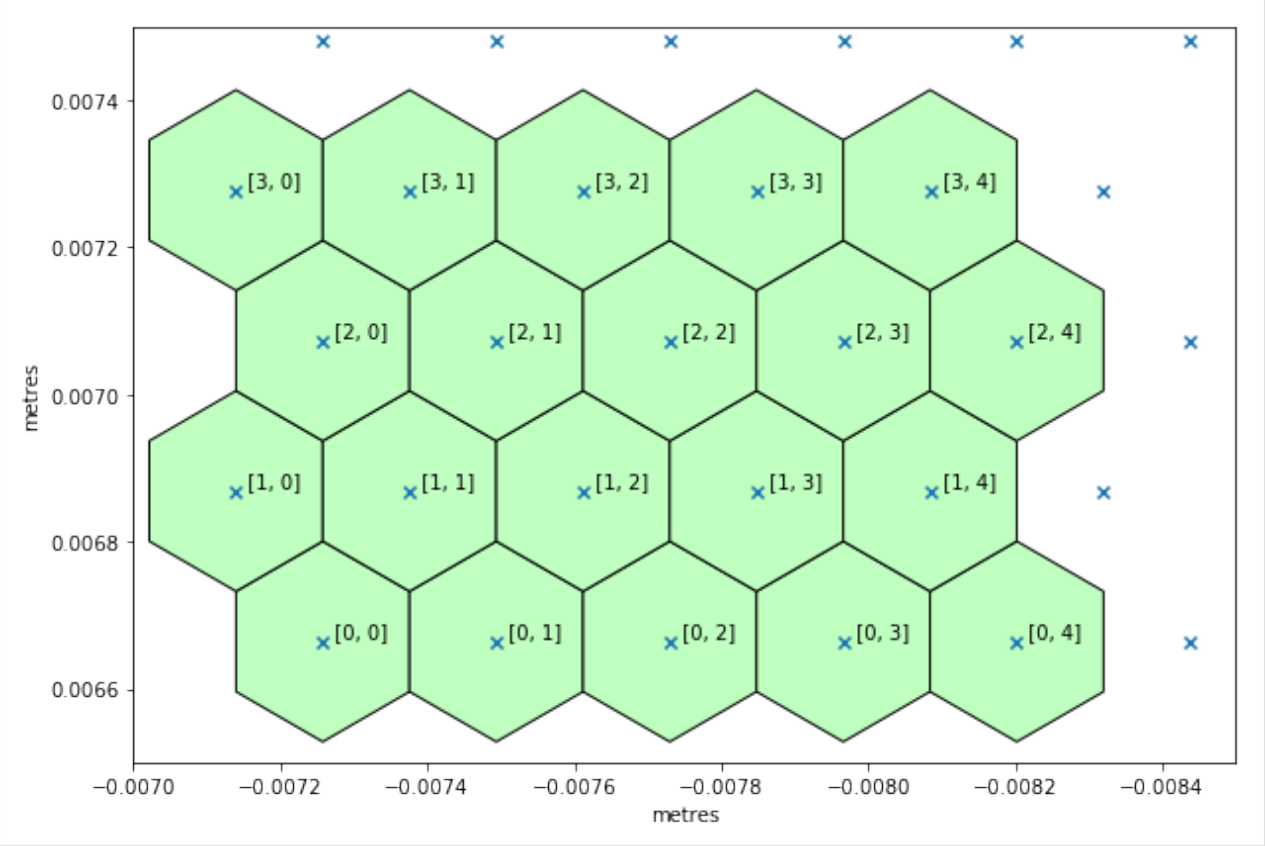
(continues on next page)

(continued from previous page)

```
q1m1_corners = distortn[:128]
```

```
Pixel positions array shape: (16, 128, 512, 3) = (modules, slow_scan, fast_scan, x/y/
↪z)
Distortion array shape: (2048, 512, 6, 3) = (modules * slow_scan, fast_scan, corners,␣
↪z/y/x)
```

[9]:
```
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection

fig, ax = plt.subplots(figsize=(10, 10))

hexes = []
for ss_pxl in range(4):
    for fs_pxl in range(5):

        # Create hexagon
        corners = q1m1_corners[ss_pxl, fs_pxl]
        corners = corners[:, 1:][:, ::-1]  # Drop z, flip x & y
        hexes.append(Polygon(corners))

        # Draw text label near the pixel centre
        ax.text(cx[ss_pxl, fs_pxl], cy[ss_pxl, fs_pxl],
                '  [{}, {}]'.format(ss_pxl, fs_pxl),
                verticalalignment='bottom', horizontalalignment='left')

# Add the hexagons to the plot
pc = PatchCollection(hexes, facecolor=(0.75, 1.0, 0.75), edgecolor='k')
ax.add_collection(pc)

# Plot the pixel centres
ax.scatter(cx[:5, :6], cy[:5, :6], marker='x')

# matplotlib is reluctant to show such a small area, so we need to set the limits␣
↪manually
ax.set_xlim(-0.007, -0.0085)  # To match the convention elsewhere, draw x right-to-
↪left
ax.set_ylim(0.0065, 0.0075)
ax.set_ylabel("metres")
ax.set_xlabel("metres")
ax.set_aspect(1)
```



3.15 Working with non-detector data

The biggest and often most important data at European XFEL comes from X-ray pixel detectors, but there are many other data sources which may be of interest. This data is often small enough to load it completely into memory, making it much easier to work with.

[1]:
```
%matplotlib inline
from karabo_data import RunDirectory
import matplotlib.pyplot as plt
import numpy as np
import re
import xarray as xr
```

3.15.1 Using pandas

This example works with data from two X-Ray Gas Monitors (XGMs). These measure properties of the X-ray beam in different parts of the tunnel. This data refers to one XGM in XTD2 and one in XTD9.

We create a pandas dataframe containing the beam x and y position at each XGM, and the photon flux. We select the columns using 'glob' patterns: $\star$ is a wildcard matching anything.

pandas makes it very convenient to work with tabular data like this, though we're limited to datasets that have a single value per train.

[2]:
```
run = RunDirectory('/gpfs/exfel/exp/SA1/201830/p900025/raw/r0150/')
```

[3]:
```
df = run.get_dataframe(fields=[("*_XGM/*", "*.i[xy]Pos"), ("*_XGM/*", "*.photonFlux
↪")])
df.head()
```

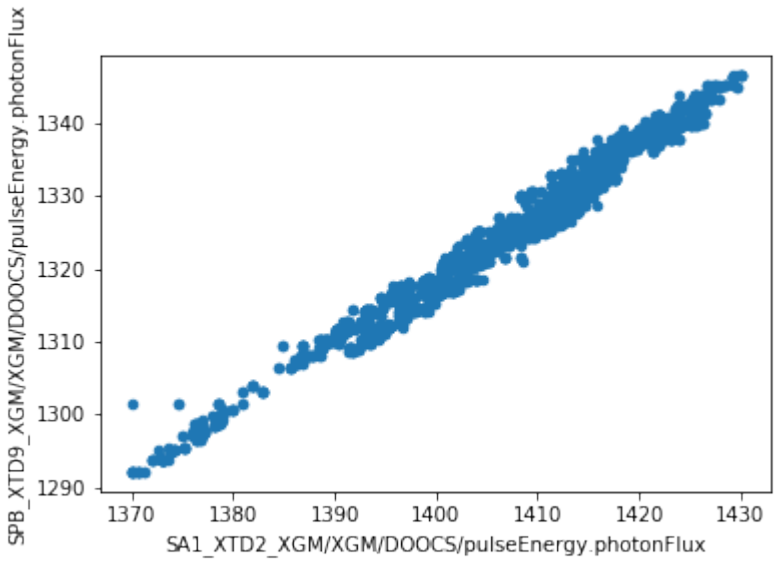
[3]:

trainId	SA1_XTD2_XGM/XGM/DOOCS/beamPosition.ixPos	SA1_XTD2_XGM/XGM/DOOCS/beamPosition.iyPos	SA1_XTD2_XGM/XGM/DOOCS/pulseEnergy.photonFlux	SPB_XTD9_XGM/XGM/DOOCS/beamPosition.ixPos	SPB_XTD9_XGM/XGM/DOOCS/beamPosition.iyPos	SPB_XTD9_XGM/XGM/DOOCS/pulseEnergy.photonFlux
142844490	2.035218	0.161399	1410.723755	-2.277912	1.717195	1327.06958
142844491	2.035218	0.161399	1410.137451	-2.277912	1.717195	1327.06958
142844492	2.035218	0.161399	1410.137451	-2.277912	1.717195	1327.06958
142844493	2.035218	0.161399	1410.137451	-2.277912	1.717195	1327.06958
142844494	2.035218	0.161399	1410.137451	-2.277912	1.717195	1327.06958

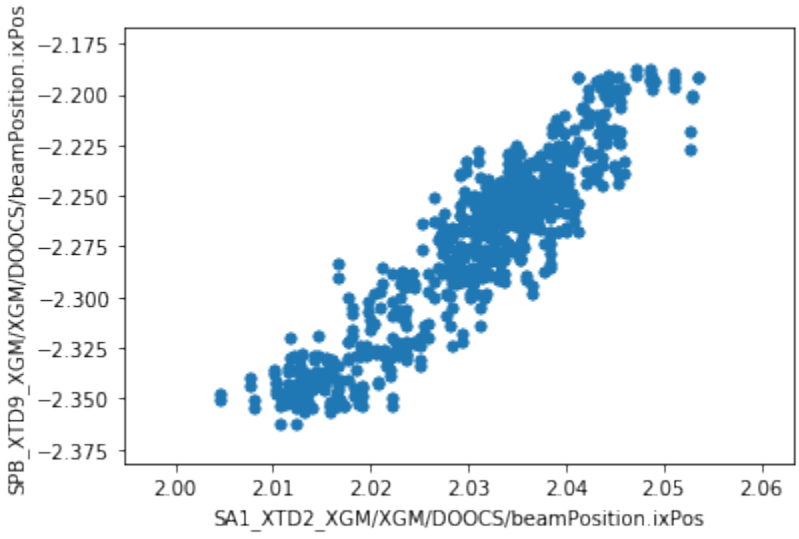
We can now make plots to compare the parameters at the two XGM positions. As expected, there's a strong correlation for each parameter.

[4]:
```
df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/pulseEnergy.photonFlux', y='SPB_XTD9_XGM/
↪XGM/DOOCS/pulseEnergy.photonFlux')
```

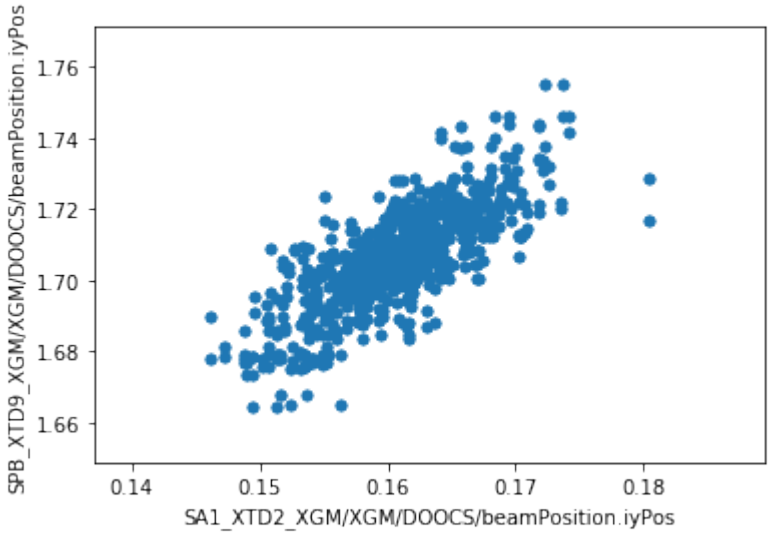
[4]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2b2de8e244a8>
```



[5]:
```
ax = df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/beamPosition.ixPos', y='SPB_XTD9_XGM/
↪XGM/DOOCS/beamPosition.ixPos')
```



[6]:
```
ay = df.plot.scatter(x='SA1_XTD2_XGM/XGM/DOOCS/beamPosition.iyPos', y='SPB_XTD9_XGM/
↪XGM/DOOCS/beamPosition.iyPos')
```



We can also export the dataframe to a CSV file - or any other format pandas supports - for further analysis with other tools.

[7]:
```
df.to_csv('xtd2_xtd9_xgm_r150.csv')
```

3.15.2 Using xarray

xarray adds pandas-style axis labelling to multidimensional numpy arrays. We can get xarray arrays for data which has multiple values per train. For example, the Photo-Electron Spectrometer (PES) is a monitoring device which records energy spectra for each train. Here's the data from one of its 16 spectrometers:

[8]:
```
run = RunDirectory('/gpfs/exfel/exp/SA3/201830/p900027/raw/r0067/')
```

[9]:
```
run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_4_A.raw.samples')
```

[9]:
```
<xarray.DataArray (trainId: 1475, dim_0: 40000)>
array([[ -6, -10,  -7, ..., -10,  -8,  -9],
       [ -8,  -8,  -7, ...,  -9,  -2, -11],
       [ -8, -10,  -7, ...,  -6,  -8, -11],
       ...,
       [ -7,  -9,  -8, ...,  -9,  -2,  -5],
       [ -5, -10,  -8, ...,  -5,  -4, -10],
       [ -7,  -8,  -7, ...,  -6,  -5,  -8]], dtype=int16)
Coordinates:
  * trainId  (trainId) uint64 128146446 128146447 128146448 128146449 ...
Dimensions without coordinates: dim_0
```

The PES consists of 16 spectrometers arranged in a circle around the beamline. We'll retrieve the data for two of these, separated by 90°. N and E refer to their positions in the circle, although these are not literally North and East.

The xarray.align() function aligns data using the axes. This is important if you're comparing data from different sources, because it matches up the train IDs. By specifying `join='inner'`, we keep only the trains which have data in both sets.

[10]:
```
data_n = run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_4_A.raw.
↪samples')
data_e = run.get_array('SA3_XTD10_PES/ADC/1:network', 'digitizers.channel_3_A.raw.
↪samples')
data_n, data_e = xr.align(data_n, data_e, join='inner')
nsamples = data_n.shape[1]
data_n.shape
```

[10]:
```
(1475, 40000)
```

We'll get a few other values from slow data to annotate the plot.

[11]:
```
# Get the first values from four channels measuring voltage
electr = run.get_dataframe([('SA3_XTD10_PES/MCPS/MPOD', 'channels.U20[0123].
↪measurementSenseVoltage')])
electr_voltages = electr.iloc[0].sort_index()
electr_voltages
```

[11]:
```
SA3_XTD10_PES/MCPS/MPOD/channels.U200.measurementSenseVoltage   -0.101792
SA3_XTD10_PES/MCPS/MPOD/channels.U201.measurementSenseVoltage   -0.111782
SA3_XTD10_PES/MCPS/MPOD/channels.U202.measurementSenseVoltage   -0.106823
SA3_XTD10_PES/MCPS/MPOD/channels.U203.measurementSenseVoltage   -0.107910
Name: 128146446, dtype: float32
```

[12]:
```
gas_interlocks = run.get_dataframe([('SA3_XTD10_PES/DCTRL/*', 'interlock.AActionState
↪')])

# Take the first row of the gas interlock data and find which gas was unlocked
row = gas_interlocks.iloc[0]
print(row)
if (row == 0).any():
    key = row[row == 0].index[0]
    target_gas = re.search(r'(XENON|KRYPTON|NITROGEN|NEON)', key).group(1).title()
else:
    target_gas = 'No gas'
```

```
SA3_XTD10_PES/DCTRL/V30300S_NITROGEN/interlock.AActionState    1
SA3_XTD10_PES/DCTRL/V30320S_KRYPTON/interlock.AActionState     1
SA3_XTD10_PES/DCTRL/V30310S_NEON/interlock.AActionState        0
SA3_XTD10_PES/DCTRL/V30330S_XENON/interlock.AActionState       1
Name: 128146446, dtype: uint32
```

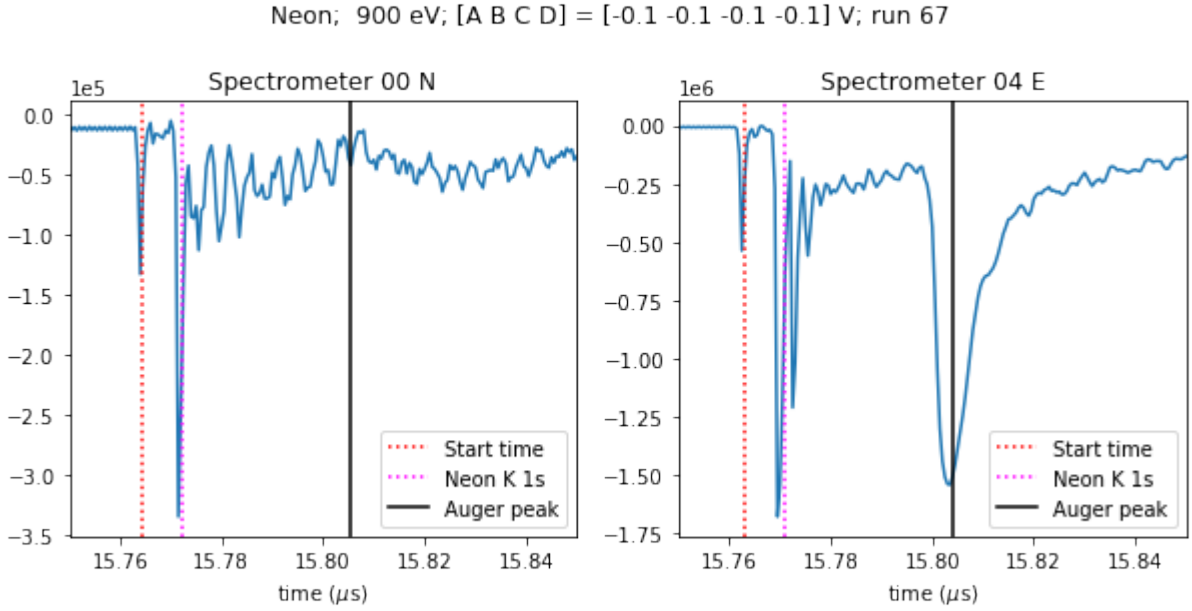
Now we can average the spectra across the trains in this run, and plot them.

[14]:
```
x = np.linspace(0, 0.0005*nsamples, nsamples, endpoint=False)

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
for ax, dataset, start_time in zip(axes, [data_n, data_e], [15.76439411, 15.
↪76289411]):
    ax.plot(x, dataset.sum(axis=0))
    ax.yaxis.major.formatter.set_powerlimits((0, 0))
    ax.set_xlim(15.75, 15.85)
    ax.set_xlabel('time ($\mu$s)')

    ax.axvline(start_time, color='red', linestyle='dotted', label='Start time')
    ax.axvline(start_time + 0.0079, color='magenta', linestyle='dotted', label='Neon␣
↪K 1s')
    ax.axvline(start_time + 0.041, color='black', label='Auger peak')
```

(continues on next page)

(continued from previous page)

```
    ax.legend()

axes[0].set_title('Spectrometer 00 N')
axes[1].set_title('Spectrometer 04 E')
fig.suptitle('{gas};  900 eV; [A B C D] = [{voltages[0]:.1f} {voltages[1]:.1f}
↪{voltages[2]:.1f} {voltages[3]:.1f}] V; run 67'
             .format(gas=target_gas, voltages=electr_voltages.values), y=1.05);
```



The spectra look different because the beam is horizontally polarised, so the E spectrometer sees a peak that the N spectrometer doesn't.

3.16 Comparing fast XGM data from two simultaneous recordings

Here we will look at XGM data that was recorded by the X-ray photon diagnostics group at the same short time interval, but at different locations of the EuXFEL-SASE. We will compare an XGM in SASE1 (XTD2) to another one in SASE3 (XTD10). These data were stored in two different runs, belonging to two different proposals even. Conceptually, this section makes use of the data-object format *xarray.DataArray*.

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

from karabo_data import RunDirectory
```

3.16.1 SASE1

Load the SASE1 run:

[2]:
```
sa1_data = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0008')
sa1_data.info()
```

```
# of trains:    6296
Duration:       0:10:29.500000
First train ID: 38227866
Last train ID:  38234161

0 detector modules ()

1 instrument sources (excluding detectors):
  - SA1_XTD2_XGM/XGM/DOOCS:output

0 control sources:
```

We are interested in fast, i.e. pulse-resolved data from the instrument source `SA1_XTD2_XGM/DOOCS:output`.

[3]:
```
sa1_data.keys_for_source('SA1_XTD2_XGM/XGM/DOOCS:output')
```

[3]:
```
{'data.intensityTD'}
```

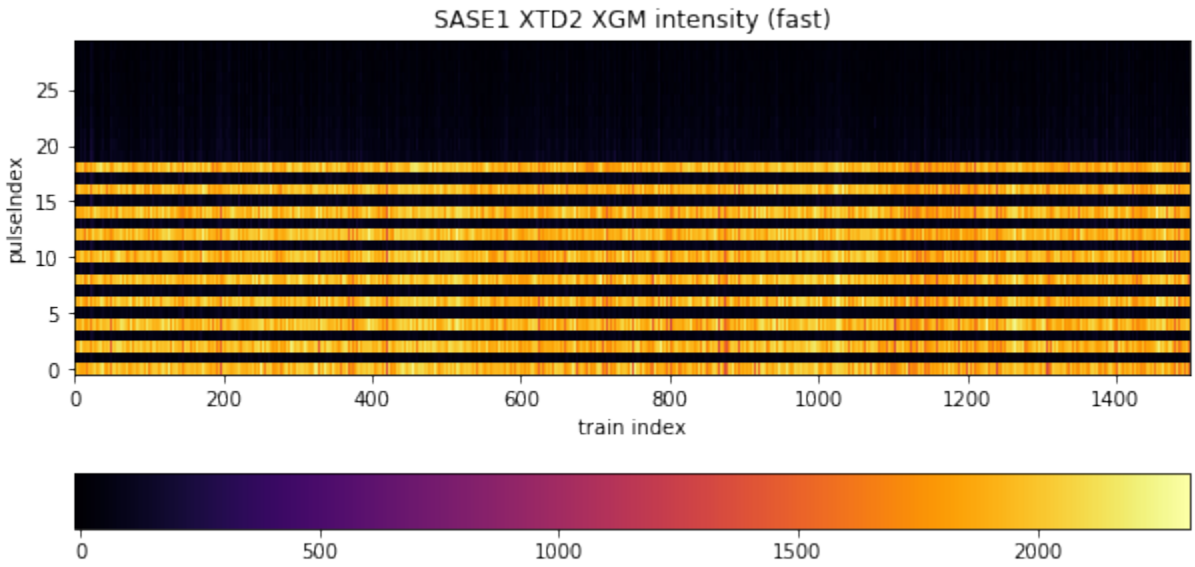
We are particularly interested in data for quantity "intensityTD". The ***xarray DataArray*** class is suited for work with axis-labeled data, and the karabo_data method `get_array()` serves the purpose of shaping a 2D array of that type from pulse-resolved data (which is originally stored "flat" in terms of pulses: there is one dimension of N(train) x N(pulse) values in HDF5, and the same number of train and pulse identifiers for reference).
The unique train identifier values are taken as coordinate values ("labels").

[4]:
```
sa1_flux = sa1_data.get_array('SA1_XTD2_XGM/XGM/DOOCS:output', 'data.intensityTD')
print(sa1_flux)
```

```
<xarray.DataArray (trainId: 6295, dim_0: 1000)>
array([[2.045129e+03, 7.820441e+01, 1.964445e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [2.091464e+03, 4.242367e+01, 1.915582e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.872965e+03, 4.368253e+01, 1.984025e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       ...,
       [1.611342e+03, 5.569377e+01, 1.811418e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.536590e+03, 6.418680e+01, 1.643087e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00],
       [1.871557e+03, 5.983860e+01, 1.738864e+03, ..., 1.000000e+00,
        1.000000e+00, 1.000000e+00]], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
Dimensions without coordinates: dim_0
```

Next, we will plot a portion of the data in two dimensions, taking the first 1500 trains for the x-Axis and the first 30 pulses per train for the y-Axis (1500, 30). Because the Matplotlib convention takes the slow axis to be *y*, we have to transpose to (30, 1500):

[5]:
```
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(1, 1, 1)
image = ax.imshow(sa1_flux[:1500, :30].transpose(), origin='lower', cmap='inferno')
ax.set_title('SASE1 XTD2 XGM intensity (fast)')
fig.colorbar(image, orientation='horizontal')
ax.set_xlabel('train index')
ax.set_ylabel('pulseIndex')
ax.set_aspect(15)
```



The pattern tells us what was done in this experiment: the lasing scheme was set to provide an alternating X-ray pulse delivery within a train, where every "even" electron bunch caused lasing in SASE1 and every "odd" bunch caused lasing in SASE3. This scheme was applied for the first 20 pulses. Therefore, we see signal only for data at even pulses here (0,2,...18), throughout all trains, of which 1500 are depicted. The intensity varies somewhat around 2000 units, but for odd pulses it is suppressed and neglegibly small.

A relevant measure to judge the efficiency of pulse suppression is the ratio of mean intensity between the odd and even set. The numpy `mean` method can work with DataArray objects and average over a specified dimension.

We make use of the numpy indexing and slicing syntax with square brackets and comma to seperate axes (dimensions). We specify `[:, :20:2]` to take every element of the slow axis (trains) and every second pulse up to but excluding # 20. That is, start:end:step = 0:20:2 (start index 0 is default, thus not put, and stop means first index beyond range). We specify `axis=1` to explicitly average over that dimension. The result is a DataArray reduced to the "trainId" dimension.

[6]:
```
sa1_mean_on = np.mean(sa1_flux[:, :20:2], axis=1)
sa1_stddev_on = np.std(sa1_flux[:, :20:2], axis=1)
print(sa1_mean_on)
```

```
<xarray.DataArray (trainId: 6295)>
array([1931.4768, 1977.8414, 1873.7828, ..., 1771.5828, 1697.2053, 1857.7439],
      dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
```

Accordingly for the odd "off" pulses:

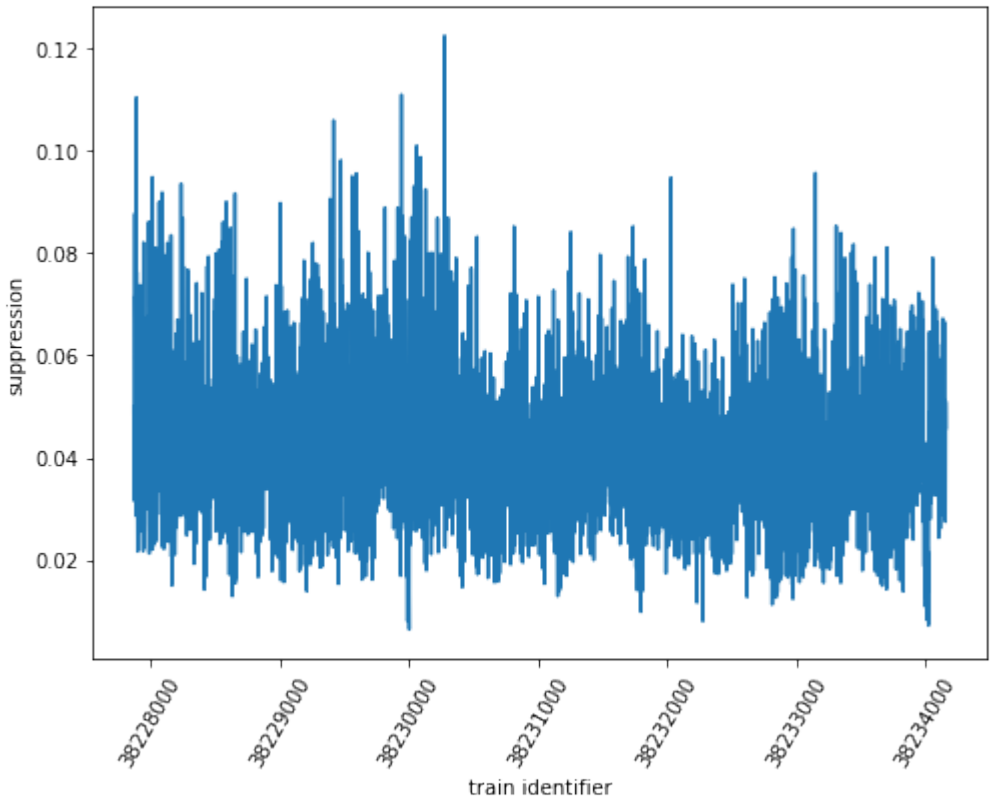
[7]:
```
sa1_mean_off = np.mean(sa1_flux[:, 1:21:2], axis=1)
sa1_stddev_off = np.std(sa1_flux[:, 1:21:2], axis=1)
print(sa1_mean_off)
```

```
<xarray.DataArray (trainId: 6295)>
array([96.10835 , 84.489044, 59.212048, ..., 90.2944  , 84.33766 , 85.03202 ],
      dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227866 38227867 38227868 ... 38234160 38234161
```

Now we can calculate the ratio of averages for every train - data types like *numpy ndarray* or *xarray DataArray* may be just divided "as such", a shortcut notation for dividing every corresponding element - and plot.

[8]:
```
sa1_suppression = sa1_mean_off / sa1_mean_on
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(sa1_suppression.coords['trainId'].values, sa1_suppression)
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

[8]:
```
Text(0, 0.5, 'suppression')
```



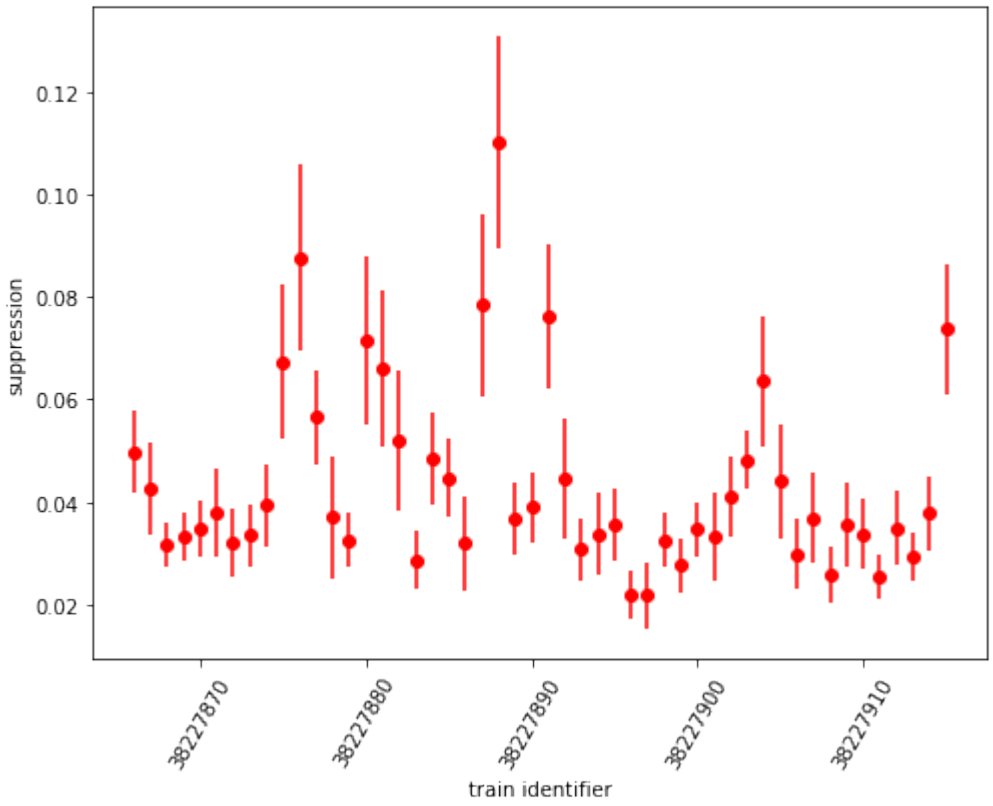
Moreover, the relative error of this ratio can be calculated by multiplicative error propagation as the square root of the sum of squared relative errors (enumerator and denominator), and from it the absolute error. The Numpy functions "sqrt" and "square" applied to array-like structures perform these operations element-wise, so the entire calculation can be conveniently done using the arrays as arguments, and we obtain individual errors for every train in the end.

[9]:
```
sa1_rel_error = np.sqrt(np.square(sa1_stddev_off / sa1_mean_off) + np.square(sa1_
↪stddev_on / sa1_mean_on))
sa1_abs_error = sa1_rel_error * sa1_suppression
```

We can as well plot the suppression ratio values with individual error bars according to the respective absolute error. Here, we restrict ourselves to the first 50 trains for clarity:

[10]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.errorbar(sa1_suppression.coords['trainId'].values[:50], sa1_suppression[:50],␣
↪yerr=sa1_abs_error[:50], fmt='ro')
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

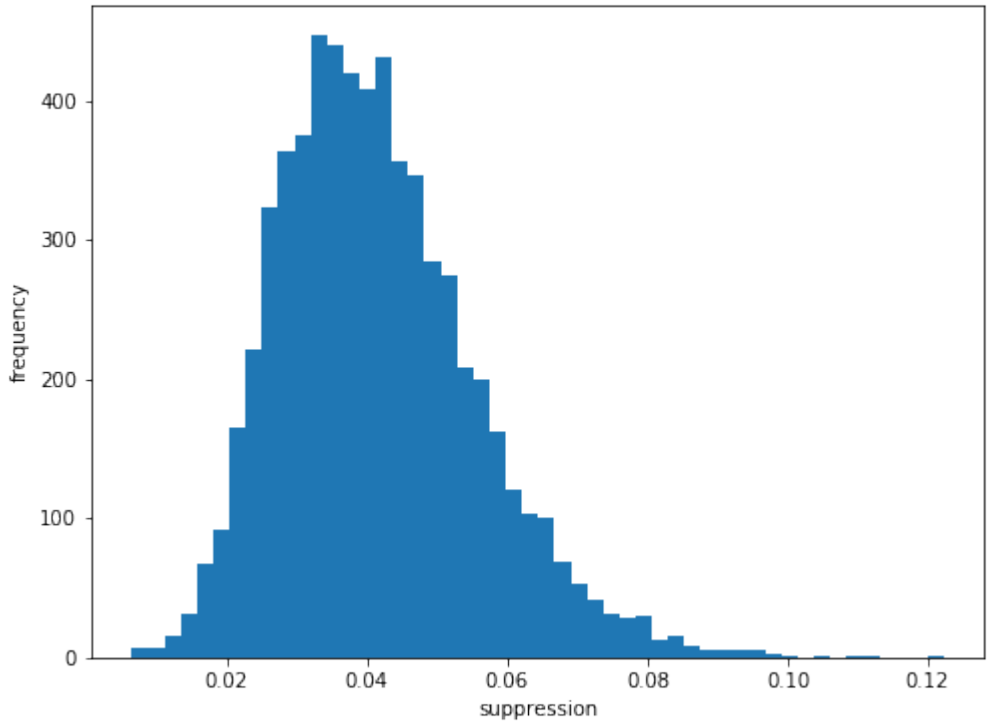
[10]:
```
Text(0, 0.5, 'suppression')
```



Finally, we draw a histogram of suppression ratio values:

[11]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
_ = ax.hist(sa1_suppression, bins=50)
ax.set_xlabel('suppression')
ax.set_ylabel('frequency')
```

[11]:
```
Text(0, 0.5, 'frequency')
```



We see that there is a suppression of signal from odd pulses to approximately 4% of the intensity of even pulses.

3.16.2 SASE3

We repeat everything for the second data set from the different run - SASE3:

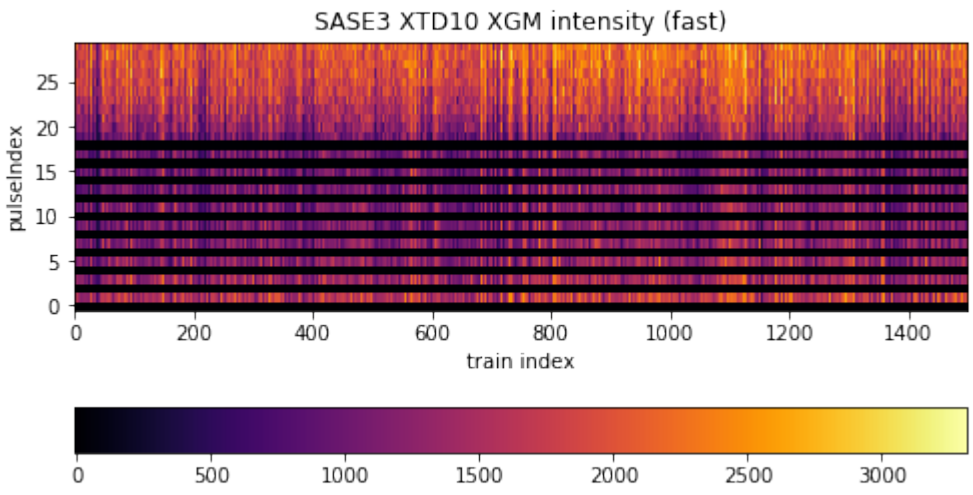
[12]:
```
sa3_data = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0009')
sa3_data.info()
```

```
# of trains:    6236
Duration:       0:10:23.500000
First train ID: 38227850
Last train ID:  38234085

0 detector modules ()

1 instrument sources (excluding detectors):
  - SA3_XTD10_XGM/XGM/DOOCS:output

0 control sources:
```

[13]:
```
sa3_flux = sa3_data.get_array('SA3_XTD10_XGM/XGM/DOOCS:output', 'data.intensityTD')
print(sa3_flux.shape)
```

```
(6235, 1000)
```

[14]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
image = ax.imshow(sa3_flux[:1500, :30].transpose(), origin='lower', cmap='inferno')
ax.set_title('SASE3 XTD10 XGM intensity (fast)')
fig.colorbar(image, orientation='horizontal')
ax.set_xlabel('train index')
ax.set_ylabel('pulseIndex')
ax.set_aspect(15)
```



The difference here is that the selection scheme (indexing and slicing) shifts by one with respect to SASE1 data: odd pulses are "on", even pulses are "off". Moreover, while the alternating scheme is upheld to pulse # 19, pulses beyond that exclusively went to SASE3. There is signal up to pulse # 70, which we could see with a wider plotting range (but not done due to emphasis on the alternation).

[15]:
```
sa3_mean_on = np.mean(sa3_flux[:, 1:21:2], axis=1)
sa3_stddev_on = np.std(sa3_flux[:, 1:21:2], axis=1)
print(sa3_mean_on)
```

```
<xarray.DataArray (trainId: 6235)>
array([ 963.89746, 1073.1758 ,  902.22656, ...,  883.9881 ,  960.5875 ,
        889.625  ], dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227850 38227851 38227852 ... 38234084 38234085
```

[16]:
```
sa3_mean_off = np.mean(sa3_flux[:, :20:2], axis=1)
sa3_stddev_off = np.std(sa3_flux[:, :20:2], axis=1)
print(sa3_mean_off)
```

```
<xarray.DataArray (trainId: 6235)>
array([5.435107, 6.615537, 8.361802, ..., 2.378666, 7.135999, 4.612433],
      dtype=float32)
Coordinates:
  * trainId  (trainId) uint64 38227850 38227851 38227852 ... 38234084 38234085
```

The suppression ratio calculation and its plot:

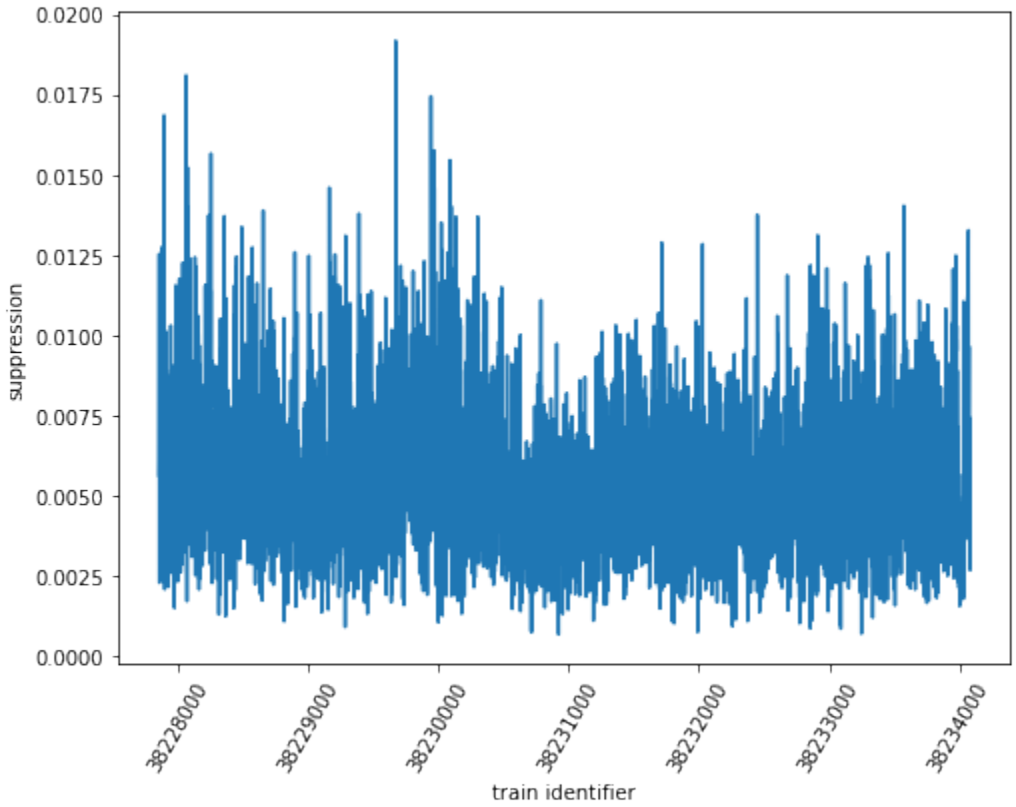
[17]:
```
sa3_suppression = sa3_mean_off / sa3_mean_on
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.plot(sa3_suppression.coords['trainId'].values, sa3_suppression)
```

(continues on next page)

(continued from previous page)

```
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

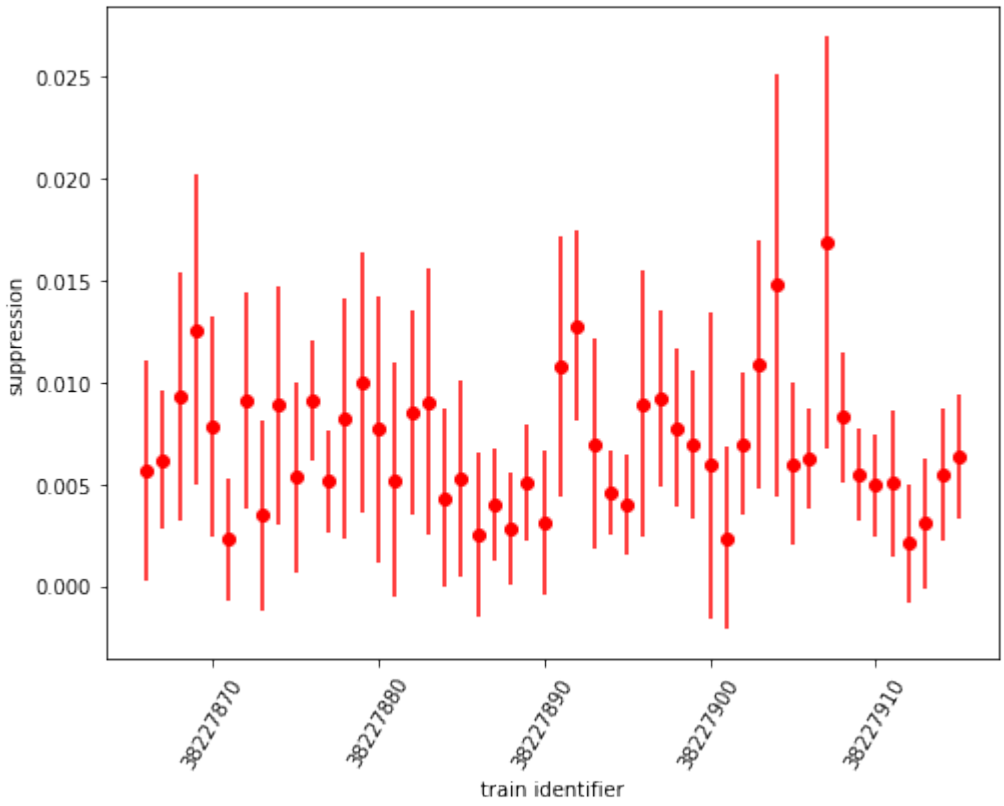
[17]:
```
Text(0, 0.5, 'suppression')
```



The error calculation with (selective) plot

[18]:
```
sa3_rel_error = np.sqrt(np.square(sa3_stddev_off / sa3_mean_off) + np.square(sa3_
↪stddev_on / sa3_mean_on))
sa3_abs_error = sa3_rel_error * sa3_suppression
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.errorbar(sa1_suppression.coords['trainId'].values[:50], sa3_suppression[:50],␣
↪yerr=sa3_abs_error[:50], fmt='ro')
ax.set_xlabel('train identifier')
ax.ticklabel_format(style='plain', useOffset=False)
plt.xticks(rotation=60)
ax.set_ylabel('suppression')
```

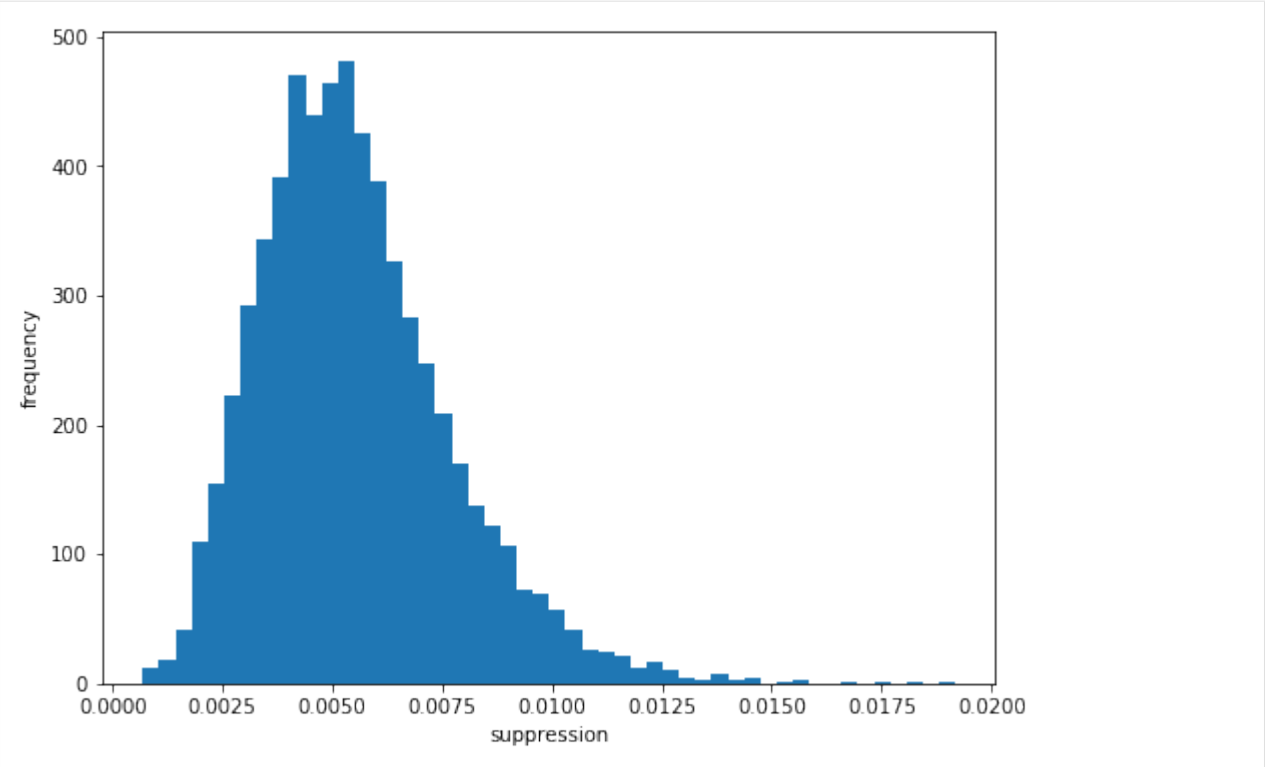
[18]:
```
Text(0, 0.5, 'suppression')
```



The histogram:

[19]:
```
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
_ = ax.hist(sa3_suppression, bins=50)
ax.set_xlabel('suppression')
ax.set_ylabel('frequency')
```

[19]:
```
Text(0, 0.5, 'frequency')
```



Here, suppression of signal for even "off" pulses is to approximately 0.5% of intensity from odd "on" pulses. The "suppression factor" is almost 10 times the value of SASE1. However, the relative error of these values is larger as well, as can be seen in the error-bar plot. For the smaller quantities, it is ~ 100% (!).

3.17 Overall comparison of suppression ratio (with error)

We ultimately want a single overall compression ratio with error for both beamlines, to complement the error-bar plots. In order to keep the error calculation simple, we do not average the mean values, but create one mean and standard deviation from a flat array of original values.

Because labeled axes are not required for this purpose, we can afford to move from the xarray.DataArray regime to Numpy array.

[20]:
```
sa1_on_all = np.array(sa1_flux[:, :20:2]).flatten()
sa1_on_all.shape
```

[20]:
```
(62950,)
```

[21]:
```
sa1_mean_on_overall = np.mean(sa1_on_all)
sa1_stddev_on_overall = np.std(sa1_on_all)
```

[22]:
```
sa1_off_all = np.array(sa1_flux[:, 1:21:2]).flatten()
sa1_off_all.shape
```

[22]:
```
(62950,)
```

[23]:
```
sa1_mean_off_overall = np.mean(sa1_off_all)
sa1_stddev_off_overall = np.std(sa1_off_all)
```

[24]:
```
sa1_suppression_overall = sa1_mean_off_overall / sa1_mean_on_overall
sa1_rel_error_overall = np.sqrt(np.square(sa1_stddev_off_overall / sa1_mean_off_
↪overall) + \
                        np.square(sa1_stddev_on_overall / sa1_mean_on_overall))
sa1_abs_error_overall = sa1_rel_error_overall * sa1_suppression_overall
print('SA1 suppression ratio =', sa1_suppression_overall, '\u00b1', sa1_abs_error_
↪overall)
```

```
SA1 suppression ratio = 0.04107769 ± 0.016009845
```

[25]:
```
sa3_on_all = np.array(sa3_flux[:, 1:21:2]).flatten()
sa3_on_all.shape
```

[25]:
```
(62350,)
```

[26]:
```
sa3_mean_on_overall = np.mean(sa3_on_all)
sa3_stddev_on_overall = np.std(sa3_on_all)
```

[27]:
```
sa3_off_all = np.array(sa3_flux[:, :20:2]).flatten()
sa3_off_all.shape
```

[27]:
```
(62350,)
```

[28]:
```
sa3_mean_off_overall = np.mean(sa3_off_all)
sa3_stddev_off_overall = np.std(sa3_off_all)
```

[29]:
```
sa3_suppression_overall = sa3_mean_off_overall / sa3_mean_on_overall
sa3_rel_error_overall = np.sqrt(np.square(sa3_stddev_off_overall / sa3_mean_off_
↪overall) + \
                        np.square(sa3_stddev_on_overall / sa3_mean_on_overall))
sa3_abs_error_overall = sa3_rel_error_overall * sa3_suppression_overall
print('SA3 suppression ratio =', sa3_suppression_overall, '\u00b1', sa3_abs_error_
↪overall)
```

```
SA3 suppression ratio = 0.005213415 ± 0.0040653846
```

3.17.1 References

1. K. Tiedtke et al., Gas-detector for X-ray lasers , J. Appl. Phys. 103, 094511 (2008) - DOI 10.1063/1.2913328
2. A. A. Sorokin et al., J. Synchrotron Rad. 26 (4), DOI 10.1107/S1600577519005174 (2019)
3. Th. Maltezopoulos et al., J. Synchrotron Rad. 26 (4), DOI 10.1107/S1600577519003795 (2019)

3.18 Parallel processing with a virtual dataset

This example demonstrates splitting up some data to be processed by several worker processes, and collecting the results back together.

For this example, we'll use data from an XGM, and find the average intensity of each pulse across all the trains in the run. This doesn't actually need parallel processing: we can easily do it directly in the notebook. But the same techniques should work with much more data and more complex calculations.

[1]:
```
from karabo_data import RunDirectory
import multiprocessing
import numpy as np
```

The data that we want is separated over these seven sequence files:

[2]:
```
!ls /gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S*.h5
```

```
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00000.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00001.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00002.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00003.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00004.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00005.h5
/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/RAW-R0034-DA01-S00006.h5
```

[3]:
```
run = RunDirectory('/gpfs/exfel/exp/XMPL/201750/p700000/raw/r0002/')
```

By making a virtual dataset, we can see the shape of it, as if it was one big numpy array:

[4]:
```
vds_filename = 'xgm_vds.h5'
xgm_vds = run.get_virtual_dataset(
    'SA1_XTD2_XGM/XGM/DOOCS:output', 'data.intensityTD',
    filename=vds_filename
)
xgm_vds
```

[4]:
```
<HDF5 dataset "intensityTD": shape (3391, 1000), type "<f4">
```

Let's read this into memory and calculate the means directly, to check our parallel calculations against. We can do this for this example because the calculation is simple and the data is small; it wouldn't be practical in real situations where parallelisation is useful.

These data are recorded in 32-bit floats, but to minimise rounding errors we'll tell numpy to give the results as 64-bit floats. Try re-running this example with 32-bit floats to see how much the results change!

[5]:
```
simple_mean = xgm_vds[:, :40].mean(axis=0, dtype=np.float64)
simple_mean.round(4)
```

[5]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ])
```

Now, we're going to define chunks of the data for each of 4 worker processes.

[6]:
```
N_proc = 4
cuts = [int(xgm_vds.shape[0] * i / N_proc) for i in range(N_proc + 1)]
chunks = list(zip(cuts[:-1], cuts[1:]))
chunks
```

[6]:
```
[(0, 847), (847, 1695), (1695, 2543), (2543, 3391)]
```

3.18.1 Using `multiprocessing`

This is the function we'll ask each worker process to run, adding up the data and returning a 1D numpy array.

We're using default arguments as a convenient way to copy the filename and the dataset path into the worker process.

[7]:
```
def sum_chunk(chunk, filename=vds_filename, ds_name=xgm_vds.name):
    start, end = chunk
    # Reopen the file in the worker process:
    import h5py, numpy
    with h5py.File(filename, 'r') as f:
        ds = f[ds_name]
        data = ds[start:end]  # Read my chunk

    return data.sum(axis=0, dtype=numpy.float64)
```

Using Python's `multiprocessing` module, we start four workers, farm the chunks out to them, and collect the results back.

[8]:
```
with multiprocessing.Pool(N_proc) as pool:
    res = pool.map(sum_chunk, chunks)
```

`res` is now a list of 4 arrays, containing the sums from each chunk. To get the mean, we'll add these up to get a grand total, and then divide by the number of trains we have data from.

[9]:
```
multiproc_mean = (np.stack(res).sum(axis=0, dtype=np.float64)[:40] / xgm_vds.shape[0])
np.testing.assert_allclose(multiproc_mean, simple_mean)

multiproc_mean.round(4)
```

[9]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ])
```

3.18.2 Using SLURM

What if we need more power? The example above is limited to one machine, but we can use SLURM to spread the work over multiple machines on the Maxwell cluster.

This is massive overkill for this example calculation - we'll only use one CPU core for a fraction of a second on each machine. But we could do something similar for a much bigger problem.

[10]:
```
from getpass import getuser
import h5py
import subprocess
```

We'll write a Python script for each worker to run. Like the `sum_chunk` function above, this reads a chunk of data from the virtual dataset and sums it along the train axis. It saves the result into another HDF5 file for us to collect.

[11]:
```
%%writefile parallel_eg_worker.py
#!/gpfs/exfel/sw/software/xfel_anaconda3/1.1/bin/python
import h5py
import numpy as np
import sys

filename = sys.argv[1]
ds_name = sys.argv[2]
chunk_start = int(sys.argv[3])
chunk_end = int(sys.argv[4])
worker_idx = sys.argv[5]

with h5py.File(filename, 'r') as f:
    ds = f[ds_name]
    data = ds[chunk_start:chunk_end]  # Read my chunk

chunk_totals = data.sum(axis=0, dtype=np.float64)

with h5py.File(f'parallel_eg_result_{worker_idx}.h5', 'w') as f:
    f['chunk_totals'] = chunk_totals
```

```
Writing parallel_eg_worker.py
```

The Maxwell cluster is divided into various partitions for different groups of users. If you're running this as an external user, comment out the 'Staff' line below.

[12]:
```
partition = 'upex'   # External users
partition = 'exfel'  # Staff
```

Now we submit 4 jobs with the `sbatch` command:

[13]:
```
for i, (start, end) in enumerate(chunks):
    cmd = ['sbatch', '-p', partition, 'parallel_eg_worker.py', vds_filename, xgm_vds.
↪name, str(start), str(end), str(i)]
    print(subprocess.check_output(cmd))
```

```
b'Submitted batch job 2631813\n'
b'Submitted batch job 2631814\n'
b'Submitted batch job 2631815\n'
b'Submitted batch job 2631816\n'
```

We can use `squeue` to monitor the jobs running. Re-run this until all the jobs have disappeared, meaning they're finished.

[14]:
```
!squeue -u {getuser()}
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

Now, so long as all the workers succeeded, we can collect the results.

If any workers failed, you'll find tracebacks in `slurm-*.out` files in the working directory.

[15]:
```
res = []

for i in range(N_proc):
    with h5py.File(f'parallel_eg_result_{i}.h5', 'r') as f:
        res.append(f['chunk_totals'][:])
```

Now `res` is once again a list of 1D numpy arrays, representing the totals from each chunk. So we can finish the calculation as in the previous section:

[16]:
```
slurm_mean = np.stack(res).sum(axis=0)[:40] / xgm_vds.shape[0]
np.testing.assert_allclose(slurm_mean, simple_mean)

slurm_mean.round(4)
```

[16]:
```
array([834.2744, 860.0754, 869.2637, 891.4351, 899.6227, 899.3759,
       900.3555, 899.1162, 898.4991, 904.4979, 910.5669, 914.1612,
       922.5737, 925.8734, 930.093 , 935.3124, 938.9643, 941.4609,
       946.1351, 950.6574, 951.855 , 954.2491, 956.6414, 957.5584,
       961.7528, 961.1457, 958.9655, 957.6415, 953.8603, 947.9236,
         0.    ,   0.    ,   0.    ,   0.    ,   0.    ,   0.    ,
         0.    ,   0.    ,   0.    ,   0.    ])
```

3.19 Averaging detector data with Dask

We often want to average large detector data across trains, keeping the pulses within each train separate, so we have an average image for pulse 0, another for pulse 1, etc.

This data may be too big to load into memory at once, but using Dask we can work with it like a numpy array. Dask takes care of splitting the job up into smaller pieces and assembling the result.

[1]:
```
from karabo_data import open_run

import dask.array as da
from dask.distributed import Client, progress
from dask_jobqueue import SLURMCluster
import numpy as np
```

First, we use Dask-Jobqueue to talk to the Maxwell cluster.

[2]:
```
partition = 'exfel'  # For EuXFEL staff
#partition = 'upex'   # For users

cluster = SLURMCluster(
    queue=partition,
    # Resources per SLURM job (per node, the way SLURM is configured on Maxwell)
    # processes=16 runs 16 Dask workers in a job, so each worker has 1 core & 16 GB␣
↪RAM.
    processes=16, cores=16, memory='256GB',
)

# Get a notbook widget showing the cluster state
cluster
```

```
VBox(children=(HTML(value='<h2>SLURMCluster</h2>'), HBox(children=(HTML(value='\n<div>
↪\n  <style scoped>\n    ...
```

[3]:
```
# Submit 2 SLURM jobs, for 32 Dask workers
cluster.scale(32)
```

If the cluster is busy, you might need to wait a while for the jobs to start. The cluster widget above will update when they're running.

Next, we'll set Dask up to use those workers:

[4]:
```
client = Client(cluster)
print("Created dask client:", client)
```

```
Created dask client: <Client: scheduler='tcp://131.169.193.102:44986' processes=32␣
↪cores=32>
```

Now Dask is ready, let's open the run we're going to operate on:

[5]:
```
run = open_run(proposal=2212, run=103)
run.info()
```

```
# of trains:    3299
Duration:       0:05:29.800000
First train ID: 517617973
Last train ID:  517621271

16 detector modules (SCS_DET_DSSC1M-1)
  e.g. module SCS_DET_DSSC1M-1 0 : 128 x 512 pixels
  75 frames per train, 247425 total frames

3 instrument sources (excluding detectors):
  - SA3_XTD10_XGM/XGM/DOOCS:output
  - SCS_BLU_XGM/XGM/DOOCS:output
  - SCS_UTC1_ADQ/ADC/1:network

20 control sources:
  - P_GATT
  - SA3_XTD10_MONO/ENC/GRATING_AX
  - SA3_XTD10_MONO/MDL/PHOTON_ENERGY
  - SA3_XTD10_MONO/MOTOR/GRATINGS_X
  - SA3_XTD10_MONO/MOTOR/GRATING_AX
  - SA3_XTD10_MONO/MOTOR/HE_PM_X
  - SA3_XTD10_MONO/MOTOR/LE_PM_X
  - SA3_XTD10_VAC/DCTRL/AR_MODE_OK
  - SA3_XTD10_VAC/DCTRL/D12_APERT_IN_OK
  - SA3_XTD10_VAC/DCTRL/D6_APERT_IN_OK
  - SA3_XTD10_VAC/DCTRL/N2_MODE_OK
  - SA3_XTD10_VAC/GAUGE/G30470D_IN
  - SA3_XTD10_VAC/GAUGE/G30480D_IN
  - SA3_XTD10_VAC/GAUGE/G30490D_IN
  - SA3_XTD10_VAC/GAUGE/G30510C
  - SA3_XTD10_XGM/XGM/DOOCS
  - SCS_BLU_XGM/XGM/DOOCS
  - SCS_RR_UTC/MDL/BUNCH_DECODER
  - SCS_RR_UTC/TSYS/TIMESERVER
  - SCS_UTC1_ADQ/ADC/1
```

We're working with data from the DSSC detector. In this run, it's recording 75 frames for each train - this is part of the info above.

Now, we'll define how we're going to average over trains for each module:

[6]:
```
def average_module(modno, run, pulses_per_train=75):
    source = f'SCS_DET_DSSC1M-1/DET/{modno}CH0:xtdf'
    counts = run.get_data_counts(source, 'image.data')

    arr = run.get_dask_array(source, 'image.data')
    # Make a new dimension for trains
```

(continues on next page)

(continued from previous page)

```
    arr_trains = arr.reshape(-1, pulses_per_train, 128, 512)
    if modno == 0:
        print("array shape:", arr.shape)  # frames, dummy, 128, 512
        print("Reshaped to:", arr_trains.shape)

    return arr_trains.mean(axis=0, dtype=np.float32)
```

[7]:
```
mod_averages = [
    average_module(i, run, pulses_per_train=75)
    for i in range(16)
]

mod_averages
```

```
array shape: (247425, 1, 128, 512)
Reshaped to: (3299, 75, 128, 512)
```

[7]:
```
[dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>,
 dask.array<mean_agg-aggregate, shape=(75, 128, 512), dtype=float32, chunksize=(75,␣
↪128, 512)>]
```

[8]:
```
# Stack the averages into a single array
all_average = da.stack(mod_averages)
all_average
```

[8]:
```
dask.array<stack, shape=(16, 75, 128, 512), dtype=float32, chunksize=(1, 75, 128,␣
↪512)>
```

Dask shows us what shape the result array will be, but so far, no real computation has happened. Now that we've

defined what we want, let's tell Dask to compute it.

This will take a minute or two. If you're running it, scroll up to the Dask cluster widget and click the status link to see what it's doing.

[9]:
```
%%time
all_average_arr = all_average.compute()  # Get a concrete numpy array for the result
```

```
CPU times: user 20.8 s, sys: 2.6 s, total: 23.4 s
Wall time: 1min 42s
```

`all_average_arr` is a regular numpy array with our results. Here are the values from the corner of module 0, frame 0:

[10]:
```
print(all_average_arr[0, 0, :5, :5])
```

```
[[48.822674 50.983025 44.953014 44.08245  45.056988]
 [45.8251   49.183388 46.39982  43.371628 47.53501 ]
 [51.03395  46.02243  44.92058  50.966656 42.918762]
 [43.190662 49.961502 44.23007  43.252197 47.663536]
 [48.844803 51.489845 50.45438  46.305546 47.51258 ]]
```

Please shut down the cluster (or scale it down to 0 workers) if you won't be using it for a while. This releases the resources for other people.

[11]:
```
client.close()
cluster.close()
```

3.20 Release Notes

3.20.1 0.7

Data access

- A new *`get_dask_array()`* method to access data as a Dask array (PR #212). Dask is a powerful tool for working with large amounts of data and doing computation in parallel.
- *`open_run()`* and *`RunDirectory()`* now take an optional `include=` glob pattern to select files to open (PR #221). This can make opening a run faster if you only need to read certain files.
- Trying to open a run directory to which you don't have read access now correctly raises PermissionError (PR #210).
- *`stack_detector_data()`* has a new parameter `real_array`. Passing `real_array=False` avoids copying the data into a temporary array on the way to assembling images with detector geometry (PR #196).
- When you open a run directory with *`open_run()`* or *`RunDirectory()`*, karabo_data tries to cache the metadata describing what data is in each file (PR #206). Once the cache is created, opening the run again should be much faster, as it only needs to open the files containing the data you want. See *Cached run data maps* for the details of how this works.
- Importing `karabo_data` is faster, as packages like xarray and pandas are now only loaded if you use the relevant methods (PR #207).
- *lsxfel* and *`info()`* are faster in some cases, as they only look in one file for the detector data shape (PR #219).
- *`get_array()`* is slightly faster, as it avoids copying data in memory unnecessarily (PR #209)

- When you select sources with *select()* or *deselect()*, the resulting DataCollection no longer keeps references to files with no selected data. This should make it easier to then combine data with *union()* in some situations (PR #202).
- *Data validation* now checks that indexes have one entry per train ID.

Detector geometry

- *plot_data_fast()* is much more flexible, e.g. if you want to add a colorbar or draw the image as part of a larger figure (PR #205). See its documentation for the new parameters.

3.20.2 0.6

Data access

- The *karabo-bridge-serve-files* command now takes `--source` and `--key` options to select data to stream. They can be used with exact source names or with glob-style patterns, e.g. `--source '*/DET/*'` (PR #183).
- Skip checking that `.h5` files in a run directory are HDF5 before trying to open them (PR #187). The error is still handled if they are not.

Detector geometry

- Assembling detector data into images can now reuse an output array - see *position_modules_fast()* and *output_array_for_position_fast()* (PR #186).
- CrystFEL format geometry files can now be written for 2D input arrays with the modules arranged along the slow-scan axis, as used by OnDA (PR #191). To do this, pass `dims=('frame', 'ss', 'fs')` to *write_crystfel_geom()*.
- The geometry code has been reworked to use metres internally (PR #193), along with other refactorings in PR #184 and PR #192. These changes should not affect the public API.

3.20.3 0.5

Data access

- New method *get_data_counts()* to find how many data points were recorded in each train for a given source and key.
- Create a virtual dataset for any single dataset with *get_virtual_dataset()* (PR #162). See *Parallel processing with a virtual dataset* for how this can be useful.
- Write a file with virtual datasets for all selected data with *write_virtual()* (PR #132).
- Data from the supported multi-module detectors (AGIPD, LPD & DSSC) can be exposed in CXI format using a virtual dataset - see *write_virtual_cxi()* (PR #150, PR #166, PR #173).
- New class `DSSC` for accessing DSSC data (PR #171).
- New function *open_run()* to access a run by proposal and run number rather than path (PR #147).
- *stack_detector_data()* now allows input data where some sources don't have the specified key (PR #141).

- Files in the new `1.0` data format can now be opened (PR #182).

Detector geometry

- New class `DSSC_Geometry` for handling DSSC detector geometry (PR #155).
- *`LPD_1MGeometry`* can now read and write CrystFEL format geometry files, and produce PyFAI distortion arrays (PR #168, PR #129).
- *`write_crystfel_geom()`* (for AGIPD and LPD geometry) now accepts various optional parameters for other details to be written into the geometry file, such as the detector distance (`clen`) and the photon energy (PR #168).
- New method *`get_pixel_positions()`* to get the physical position of every pixel in a detector, for all of AGIPD, LPD and DSSC (PR #142).
- New method *`data_coords_to_positions()`* to convert data array coordinates to physical positions, for AGIPD and LPD (PR #142).

3.20.4 0.4

- Python 3.5 is now the minimum required version.
- Fix compatibility with numpy 1.14 (the version installed in Anaconda on the Maxwell cluster).
- Better error message from *`stack_detector_data()`* when passed non-detector data.

3.20.5 0.3

New features:

- New interfaces for working with *AGIPD, LPD & DSSC Geometry*.
- New interfaces for accessing *AGIPD, LPD & DSSC data*.
- *`select_trains()`* can now select arbitrary specified trains, not just a slice.
- *`get_array()`* can take a region of interest (`roi`) parameter to select a slice of data from each train.
- A newly public *`keys_for_source()`* method to list keys for a given source.

Fixes:

- *`stack_detector_data()`* can handle missing detector modules.
- Source sets have been changed to frozen sets. Use *`select()`* to choose a subset of sources.
- *`get_array()`* now only loads the data for selected trains.
- *`get_array()`* works with data recorded more than once per train.

3.20.6 0.2

- New command `karabo-data-validate` to check the integrity of data files.
- New methods to select a subset of data: *`select()`*, *`deselect()`*, *`select_trains()`*, *`union()`*,
- Selected data can be written back to a new HDF5 file with *`write()`*.
- *`RunDirectory()`* and *`H5File()`* are now functions which return a `DataCollection` object, rather than separate classes. Most code using these should still work, but checking the type with e.g. `isinstance()` may break.

See also:

Data Analysis at European XFEL

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

C

D

F

G

H

I

K

L

O

P

R

S

T

U

W

Z